ConTExt User Guide

Michael Peter McGurk

May 19, 2017

Contents

1 Overview

Repeat-derived NGS reads are often ignored in sequencing analyses because an essential step in most NGS pipelines is identifying the genomic locus from which a given read originates. For reads originating in repetitive sequence, this is often impossible. Consequently, asking questions about repeats using NGS data requires phrasing the questions differently than one would for unique sequence. ConTExt is a set of tools designed to do just that. Instead of trying to identify the locus from which a repeat-derived read originates, it seeks to determine the repeat-family from which the read originates and the location within the repeat-family (e.g. for a TE-derived read, does it sit near the 5' end of the element or the nearer to the 3' end?). Once reads are organized in this manner, ConTExt makes use of read pair information, coverage, and sequence polymorphism to identify structures involving repetitive sequence (tandem junctions, deletions within a repeat, insertions into unique or repeated sequence), estimate their copy number, and identify the proportion. These structures can be examined for population variation.

A guiding tenet of ConTExt is that visualizing data is good practice. Tables are useful for crunching numbers, but visually exploring data is often essential to understanding and trusting it. Consequently, we organize repeat-derived reads in a manner conducive to understandable visualizations. To this end, ConTExt ultimately aligns repeat-derived reads to the consensus sequences of known repeat families. It is much easier to make inferences about the structures formed by Hobo transposon when all of the Hobo-derived reads are gathered in one place and organized with respect to a single Hobo consensus sequence. However, aligning directly to consensus sequences is likely to miss reads originating from old, divergent copies of a repeat. To get around this, we use a two-step procedure wherein reads are first aligned to the set of all individual instances of repeats identified in the reference genome, and then these alignments are collapsed onto the RepBase consensus sequences. The first step provides power to detect divergent elements, and the second ensures ease of visualization and analysis. In D. melanogaster, we found that this two-step procedure identified about 25% more repeat-derived reads than did aligning directly to the consensus sequences.

Once the data are aligned, ConTExt utilizes two-dimensional representations of the alignments. Put simply, because each read pair contains two reads, each of which aligns to a different position, each read pair can be thought of as a point in a 2D scatter plot. A variety of summaries are computed for each data set, including the distributions of mate pair distances and the read depth distributions conditioned on %GC. Structures involving repeats are identified by fitting a generative model to the alignments using Expectation Maximization.

Most of the major scripts that actually organize the data are designed to be run from command-prompt and all of the necessary steps are called automatically. The individual functions are not intended to be called directly by the user. Extensive documentation for using functions is restricted to those intended to be called manually.

We automate as much of the pipeline as is possible, but because repetitive DNA is the red-headed child of bioinformatics cookie cutter solutions do always not exist for every step of analysis. Where user decision are necessary, this manual provides guidance and we have included some tools that made our lives easier–hopefully they will do the same for yours.

This user guide is arrange in the

This module contains takes an input directory and

2 Preparing the repeat index and reference genome

2.1 Preparing the repeat consensus index

Lists of repeat consensus sequences can be obtained from a number of sources, the most comprehensive of which is RepBase. We find it useful to supplement this with sequences for interesting multigene families, like the ribosomal RNA genes and the tandemly arrayed (in Drosophila) histone genes. Some exploration of the literature may be useful when constructing a good repeat index.

However, as useful as RepBase libraries are, they often include redundant sequences and multiple sub-families of individual repeat families. These can greatly complicate downstream analyses with ConTExt, and need to be removed. Manually curating a *.fasta containing hundreds of entries is not a trivial task. To simplify it, we provide a submodule called FindHomology which carries out all pairwise alignments within your index of repeat consensus sequences and then uses the Louvain community finding algorithm to identify groups of potentially redundant entries. This greatly reduces the effort required to obtain a suitable repeat index.

FindHomology first uses BLASTn to carry out all pairwise alignments. It then represents these alignments as a graph, which is a structure useful for representing relationships between objects. In this case, the objects are seqeunces and the relationships are alignments. Therefore, we represent each sequence in the repeat index with a node and we connect a pair of sequences with an edge if BLASTn identified an alignment between them with greater than 80% identity. Groups of sequences that tend to align to each other create communities in this graph, which are readily identified by the Louvain algorithm.

We do not implement the Louvain algorithm ourselves, but rather make use of Thomas Aynaud's easy to use implementation. This dependency is not available through PIP but can be found at https://github.com/taynaud/python-louvain/. Installation is as easy as running the module's setup.py.

Additional dependencies for FindHomology are Biopython and Networkx, both of which can be installed with PIP.

To run the script from command-line:

python FindHomology.py -B [Path to Blast directory] -i [Path to repeat index] -o [Path to output directory]

For example:

python FindHomology.py -B "C:/Program Files/NCBI/blast-2.6.0+" -i c:/barbashlab/Repbase_19.06_DM_Curated_5-1-15_sim.fa -o c:/barbashlab/find_homology_test

Note: This script creates a temporary folder called /temp in the parent of the output directory. Make sure no such folder already exists.

The output folder will contain a few different files:

- _err.txt
- alignment_output.tsv
- alignments_<##>_<##>.tsv
- output.csv
- communities.txt
- community_<##>.fa

The interesting files are communities.txt and the community_<#>.fa. Communities.txt lists all of the communities identified and the names of repeat entries comprising each community. Removing redundant entries is simply a matter of going through each community and removing any entry that contains significant homology with another entry. In the extreme case, this means paring each community down to a single entry. In practice, this isn't always necessary and some personal judgement is required here. To aid in this, each community is assigned a number and has a corresponding community_<#>.fa file which contains the sequences of all repeats that comprise the community. Aligning the community_<#>.fa against itself provides insight into the nature of the homologies that create the cluster. In some cases, the community may by driven by a short tract of sequence with low (<85%) homology. In this case, it may be fine to leave both entries in the index; though, be sure to remember that this homology exists if you happen to see some interesting signal involving one of these elements in downstream analyses. Often, communities contain a full-length element along with internally deleted, fragmented, or variant elements. In this case it makes sense to retain only one full-length element and it can be useful to align the community_<#>.fa against the reference genome to get a sense of which entry appears most representative. Remember, though, non-autonomous DNA transposons are often internally deleted and abundant, so the most abundant variant isn't necessarily the best. Looking for ORFs within repeat entries may help resolve ambiguities here. In others cases, a community may be comprised of many variants of a very diverse repeat family. In this case, we favor examining all pairwise alignments in the web BLAST utility and retaining a few entries based on the distance tree. The most irritating instances are composite repeats, like the Suppressor of Stellate sequence in D. melanogaster, which contains a subsequence of a tandemly repeated gene (Stellate) flanked by the ends of a DNA transposon (Protop). To resolve this, we discarded the Su(Ste) entry, and simply retained the Protop and Stellate entries. This seems unsatisfying, but it is the best that can be done: Given our alignment pipeline, there is no

Remember: The index should contain one good representative of each repeat family you are interested in. It doesn't have to perfect though, because it will be fed through RepeatMasker to identify many instances of these repeat families in the reference genome.

The remaining files are _err.txt, which is an error log created by BLAST and should be empty (we hope...). The Blast outputs are stored in alignment_output.tsv, alignments_<#>_<#>.tsv, output.csv.

Once you're satistfied with your curation, run the curated repeat index through FindHomology again to make sure there aren't any communities that were missed the first time. If not, you're ready to mask the genome.

2.2 Preparing the reference genome

Before moving forward, it may be necessary to modify the reference genome a bit. Because ConTExt produces a separate file for each entry in the reference genome and repeat consensus index, things get messy if the reference has a separate entry for each unmapped contig or scaffold. Even well curated assemblies, like D. melanogaster's, have thousands of short, unmapped contigs. These are mostly repetitive, and actually quite useful for our purposes (we'll get to that). However, they need to be concatenated into a single sequence entry. We provide a few scripts for doing that.

[...talk about these scripts...]

Once the repeat consensus index is constructed, it is used to mask the reference genome to which the reads will be aligned. RepeatMasker can be somewhat tricky to install. It requires a Linux operating system (though you might be able to manage in a virtual environment, like VirtualBox). It's also necessary to modify the permissions of the executables to allow them to execute. Once RepeatMasker is installed, you can mask the reference as follows:

RepeatMasker -s -lib <path to consensus sequences> -dir <output directory> -pa 3 <reference genome>

2.3 Obtaining the index of individual repeats

The RepeatMasker output includes a few files, including the masked genome and a *.ori.out file. The *.out file contains the coordinates of each masked repeat, which we will use to finish our repeat index and construct the table needed to collapse alignments onto the repeat consensus sequences. First, however, we need to convert the *.out to a *.gff3 using one of RepeatMasker's utilities:

perl utils/rmOutToGFF3.pl *.out > *.gff

You can tell RepeatMasker to generate a *.gff when it masks the genome, but the format of this differs from that expected by the next few steps, so running rmOutToGFF3 is neccesary.

The next step requires calling AlignmentConverter.py from command prompt. Here, the *.gff, the repeat consensus index, and the unmasked reference genome are used to generate both the final repeat index and the conversion table as follows:

python [path to AlignmentConverter.py] -fxn build -cons [path to FASTA of repeat consensus sequences] -ref [path to FASTA containing the unmasked reference genome] -gff [path to a *.gff3 obtained using RepeatMasker indicating where individual repeats are in the reference] -out [path to your output file] -BLAST [path to BLAST]

This command will both create a tab-delimited file that stores the conversion table and a *.fasta containing both all of the individual repeats extracted from the genome and consensus sequences.

2.4 Generating the Bowtie2 indexes

The *.fasta created by AlignmentConverter will be the repeat index to which reads are initially aligned and the masked reference genome will be the assembly to which reads are aligned. These need to be converted to BowTie2 indices. Refer to the Bowtie2 manual for details on how to do this (it's quite easy).

3 Aligning the data

The first major step of ConTExt is aligning repeat-derived reads to repeat consensus sequences. Because ConTExt is intended to be used with large numbers of samples, the alignment pipeline is automated in ConTExt.py, which is intended to be run from command line—the internal functions are not general enough to be useful without modification. The general

1. Identifies all *.fastq files in the input directory corresponding to paired end data
2. Iterates over each paired end sample and...
   (a) Carries out pre-alignment processing with Trimmomatic (adapter removal, quality trimming, removal of very short reads)
   (b) Trims reads from their 3'-ends to increase the mate-pair distance (gap size)
   (c) Aligns reads to both the masked reference genome and a set of repeat sequences
   (d) Collapses repeat alignments onto the corresponding consensus sequence (using a separate script: Alignment_Converter.py)
   (e) Matches paired reads
   (f) Creates a new directory containing all of the output files for the sample

3.1 Setting up the input directory

ConTExt was designed to to work with population genomic data. To simplify working with many datasets, ConTExt isn't run on a single paired-end dataset at a time, but rather on a directory which contains at least one (and possibly hundreds) paired-end datasets. So to begin, place all of the samples you wish to work with in a single directory. It doesn't matter if there are extraneous files in the directory—they will be ignored.

Two important points:

1. For each sample, the forward and reverse reads must be split into separate files, flagged with "_1" for forward reads and "_2" for reverse reads. If a sample does not have two such files, it will be ignored by ConTExt. If you are working with data downloaded from the Short Read Archive (SRA), splitting forward and reverse reads into separate files can be accomplished with SraToolkit's FastqDump by specifying –split-3 at command-line.
2. The function that identifies samples expects input files to be named as follows:

   <sample_name>_{1/2}.{fastq/fq}(.gz)

   where

   <...> Any string (can include underscores)
   (...) indicates something which is optional
   {.../...} indicates allowed values

3.2 The configuration file

In many ways, the module ConTExt.py is Python code masquerading as a BASH script, in that it makes a number of calls to external programs with the subprocess module. Aligning the data requires telling ConTExt where it can find these programs. Rather than make the necessary command messier than it already is, ConTExt requires the path to a configuration file be passed as a commandline parameter.

This configuration file should be a text file with three lines, formatted exactly as follows:

```
B=<path to the Bowtie2 directory>
T=<path to the Trimmomatic executable>
S=<path to the directory containing the SamTools executable>
```

As an example, this is the config file I use:

```
B=/home/mpm289/bowtie2-2.1.0
T=/home/mpm289/Trimmomatic-0.33/trimmomatic-0.33.jar
S=/bin
```

3.3 Conversion Table

This is the tab-delimited table necessary for collapsing alignments onto the corresponding consensus sequences, as described in 2.3. It is passed to the -conv parameter.

3.4 Consensus File

Collapsing alignments on to the consensus sequences requires the original index of repeat consensus sequences. So the path to the index of repeat consensus sequences, as described in 2.1, must be passed to the -cons parameter

3.5 Trimming reads to increase the mate-pair distance

Ultimately the aligned data will be used to identify structures formed by repetitive sequence, which can be used as structures. These structures are identified by looking for read pairs spanning junctions, sequence coordinates that are not adjacent in the reference sequences, but which are adjacent in the sequenced genome (e.g. deletion breakpoints, the boundary between two tandemly arrayed elements). Such junctions are only identified when they fall between the two reads in a read pair, and are missed when they are contained in a sequencing read (such a read will not align). To increase our power to identify junctions, we employ a trick—we trim sequence from the 3' ends of reads, which shortens the reads but increases the mate pair distance.

The maximum read length allowed is specified by the parameter -L, and any reads longer than this value are trimmed from their 3' ends. For the Global Diversity Lines, where the average read length is 100nt, we set -L to 70.

3.6 Running the alignment

As noted before, the alignment pipeline is called through command-line. The

```
nohup python conTExt_linux_34.py -config <path> -i <path> -o <path>
-t <path> -a <path> -conv <path> -cons <path> -L <int> -p <int> –phred
{33/64} -R {True/False} -G {True/False}
```

Parameter	Description
-config	Path to a text file that specifies the paths to Bowtie2, SamTools, and Trimmomatic
-i	Path to the input directory
-o	Path to the output directory (will be created if it does not already exist)
-R	Create a copy of the *.fastq wherein reads are renamed to ensure they are compatible with the pipeline. Does not modify the original files. (True/False; if in doubt: True)
-L	Maximum desired read length
-t	Path to the Bowtie2 index of repeat sequences (stripped of any file extensions)
-a	Path to the Bowtie2 index of the repeat masked reference genome (stripped of any file extensions)
-cons	Path to *.fasta containing the repeat consensus sequences
-conv	Path to table used to convert alignments from individual insertions to repeat consensus sequences
-p	Number of processors to use
–phred	PHRED encoding of the data (33 or 64)
-G	Whether data has GemCode barcodes (True/False)

Because running dozens to hundreds of alignments can take days or weeks, we recommend disconnecting the process from terminal using nohup. All of the external programs called can make use of multiple processors, so if you have access to multiple CPUs, setting -p >1 will speed up this step. For the sake of your computer, though, we do not recommend using all available processing power; we never commit more than half of the available CPUs to lengthy procedures like alignments.

Sample command:

```
nohup python conTExt_linux_34.py -i "/data2/mpm289/iso-1_fastq"
-o "/data2/mpm289/iso1_output" -t "/home/mpm289/bowtie2-2.1.0/ins_and_cons_index_7_8_16"
-a "/home/mpm289/bowtie2-2.1.0/dmel-r6.01_masked_7_8_16" -p 6 -L 70 -
conv "/data2/mpm289/ReferenceGenomes/Dmel_reference/ins_and_cons_index_7_8_16.tsv"
–phred 33 -cons "/data2/mpm289/ReferenceGenomes/Repbase_19.06_DM_7-
8-16.fa" -G False -R True -config /home/mpm289/config.cgi > "/data2/mpm289/iso-
1_fastq/nohup_context.txt"
```

3.7 Output

The following output files will be generated for each sample <sample>:

1. ./<sample>_log.txt
2. ./<sample>_{1/2}_{as/te}_bowtie_log.txt (4 files in total)
3. ./<sample>_errlog.txt
4. ./<sample>/
    - Repeat1.dat
    - Repeat2.dat
    - ...
    - Chr1.dat
    - Chr2.dat
    - ...

<sample>_log.txt is the Trimmomatic output. The four <sample>_{1/2}_{as/te}_bowtie_log.tx files contain the Bowtie2 alignment statistics. <sample>_errlog.txt is mostly useless and is description how Samtools manipulated the aligned data. In addition, once the entire input directory is processed, a runlog.txt file is created. This contains the contents of sample-specific log files and some additional information. Importantly, at the end of runlog.txt is a list of the samples that were successfully processed, as well as any that samples where ConTExt failed.

You will note that no *.sam or *.bam files are included in the output. These are generated during the pipeline, but are subsequently deleted to save space. The alignments are instead organized in custom tables that simplifies the downstream clustering. These tables are created in a subdirectory named after the sample (./<sample>/). Any read pair where at least one end is anchored in Repeat1 is written as a row in ./<sample>/Repeat1.dat. Any read pair where at least one end is anchored in Chr1 (but no end is anchored in a repeat) is written as a row in ./<sample>/Chr1.dat. Read pairs where both ends are unmapped are discarded.

The columns of these files are as follows:

Row	Description
1	Reference sequence to which Read 1 aligns
2	Strand to which Read 1 aligns
3	Start coordinate of Read 1 alignment
4	End coordinate of Read 1 alignment
5	Sequence of Read 1
6	Quality string of Read 1
7	Reference sequence to which Read 2 aligns
8	Strand to which Read 2 aligns
9	Start coordinate of Read 2 alignment
10	End coordinate of Read 2 alignment
11	Sequence of Read 2
12	Quality string of Read 2
13	CIGAR String (Read 1)
14	CIGAR String (Read 2)
15	MAPQ (Read 1)
16	MAPQ (Read 2)
17	SNP string (Read 1)
18	SNP string (Read 2)

3.8 Alternatives

The above alignment pipeline is geared toward finding structures within repeats. If you are simply interested in assigning repeat-derived reads to the corresponding consensus sequences, you can take an alternative approach. First using BowTie2, align your sequencing reads to the *.fasta described in 2.3; the output of the alignment step needs to be a *.sam rather than a *.bam

Then in command prompt use AlignmentConverter to collapse the alignments onto the consensus:

python [path to AlignmentConverter.py] -fxn convert -i [path to *.sam file] -conv [path to conversion table]

3.9 General tips

- Use nohup to disconnect process from terminal
-

4 Identifying structures

In the last step we aligned reads, now we need to think about the patterns in those alignments to identify structures in repetitive sequence. Our approach is to find a generative model that well-describes the alignment patterns in a NGS data set and which relates these to the underlying structures. Each structure in a sequenced genome generates a distinct distribution of discordant read pairs. The distribution of all discordant read pairs in a data set can then be thought of as a mixture of the individual distributions produced by different structures. So, we approach this problem using the well-founded toolbox of Mixture Modeling. The actual generative model is difficult to fit, so we employ an easy to solve approximation and model the discordant read pairs as arising from a Gaussian Mixture Model (GMM) We fit the GMM to the data using Expectation Maximization and use the fitted model to identify structures.

Accomplishing this is entirely automated.

4.1 Running ExpMax.py

The ge

OMP_NUM_THREADS=1 nohup python ExpMax.py -i <path> -ref <path> "/data2/mpm289/References_RepeatMasker_Troubleshooting/dmel-r6.01_concatenated.fa" -o <path> -cons <path> -f dir -d cat -head <string> -x <int> -y <int> -a 'x'

If you wish to process only a subset of samples in the input directory, we provide the optional parameters -head <char> and -find <string>. -head tells ExpMax to only process those samples that begin with the specified character. -find tells ExpMax to only process those samples that contain the specified string. Neither of these parameters are case-sensitive. As ExpMax does not These functions are particularly useful in cutting down the amount of time it takes to cluster a population dataset

As the EM implementation is quite fast, most of the functions do not make use of parallel processing. if you have access to multiple CPUs, you may

We recommend explicitly limiting the process to a single CPU with OMP_NUM_THREADS=1. While the vast majority of the script runs on a single-processor, some of the numerical operations However we do call Scikit-Learn's Gaussian Process implementation, which will use as much processing power as it can. Because we like to

4.2 Output

ExpMax.py generates a number of output files.

File Name	Description
<sample>_len.tsv	The read length distribution of the processed data.
<sample>_kde.tsv	The mate pair distance (MPD) distribution. There are four rows. The first is the observed frequency art which a given MPD is observed, the second is the kernel density estimate of the underlying distribution, and the third is the expected distribution conditioned on the read pair spanning a junction. The fourth row is the corresponding MPDs; negative values indicate overlapping reads.
<sample>_cvg_exp_{chromosome}.tsv	The average depth on a given chromosome as a function %GC.
<sample>_cvg_hist.npy	A Numpy array that stores the joint distribution of read depth and %GC for each chromosome. This is used downstream to estimate the copy numbers of structures.
<sample>_cvg_GCcounts.npy	The distribution of %GC across the reference genome.

5 Matching junctions between samples

5.1 Manually

5.2 Automated — To implement

Automatically matching junctions between samples also appears to be a Mixture Modeling problem, though the structure of the problem is not well suited to EM. Because we estimate junction location using the mean position of read pairs spanning the junction, our location estimates should arise from normal distributions centered on the true junction (c.f. the Central Limit Theorem). The error of our estimate depends on both the read count of the structure in that sample and the MPD distribution of the sequencing library. Thus, the distribution of junction estimates arises from a mixture of many component distributions, each of which corresponds to a true junction. However, while these component distributions have location parameters, the shape of clusters is driven not by a covariance parameter associated with each true junction, but rather an error associated with each junction estimate. Intuitively, this seems sufficient information to estimate the location parameters, and we suspect a k-Means algorithm with soft cluster assignments based on model likelihood should do a reasonable job of matching junctions between samples and automatically choose the right number of clusters.

6 Sequence Analysis

One of the important data structure ConTExt can produce is a SNP pileup for either a repeat consensus sequence, or for positions flanking a structure, such as tandem junction. This data structure allows sequence variation in to be explored using standard population genetic tools and summaries.

The pileup is structured in a 3D numpy array. The axes of the array are as follows

Axis	Description
0	Sample
1	Allele. There are five possible alleles, which are index as follows—Consensus allele: 0, A:1, T:2, C:3,G:4
2	Consensus position.

The value, X, stored at [i,j,k] in the table can be read as follows "In sample i at position k of the consensus there are X reads with PhredQ>=30 that support allele j". The samples are indexed in alphabetical order.

To load this array:

rpt_array=numpy.load(path to file)

Functions for interpretin this array are found in the PCA_Analysis module. To load:

from PCA_Analysis import *

This module has the following dependencies:

Scikit-learn .18 Statsmodels .8 xlsxwriter matplotlib numpy scipy

6.1 Exploring the SNP array

Prior to moving forward it may be useful to examine the array and assess the read depth across the consensus sequence.

To visualize the average read depth at each position:

from matplotlib import pyplot

pyplot.plot(numpy.mean(rpt_array.sum(1), ax=0))

pyplot.show

Setting ax=0 tells numpy to average across the first axis only, which corresponds to averaging across samples.

To visualize the minimum read depth at each position:

pyplot.plot(numpy.min(rpt_array.sum(1), ax=0))

pyplot.show

Knowing the minimum is important as PCA does not handle missing data well. For PCA it is necessary to either impute the missing values, or exclude positions with missing values. I take the latter route–that is, I require I having information about a position in every sample.

6.2 Determining major allele proportion

Because repeats are present in multiple copies per genome, one approach to considering the homogeneity of a repeat family is to ask what fraction of copies contain a particular allele.

This can be accomplished with the follow command:

rpt_prop, rpt_freq, rpt_pos=ComputeMajorAlleleStatistics(rpt_array)

This function computes two statistics from the SNP pileup.

First, it defines the major allele at each positions as the allele present at the highest average allele proportion across the populations.

Second, it estimates the proportion of samples that contain a variant allele at a position in at least one repeat unit. Variant alleles are called by rejecting the null hypothesis that all observed variants are sequencing errors using a conservative error rate ϵ. This does not account for errors introduced by PCR, which may be important. For each position i in sample j total read count X_{ij} and variant read count k_{ij} are determined. We define the variant allele as the allele with the second greatest average allele proportion. The null model is:

$k_{ij} \sim Binom(X_{ij}, \epsilon)$

A p-value is computed for each sample and the null hypothesis is (or isn't) rejected using the Benjami-Hochsberg multiple testing correction for the number of samples at the given FDR (default .01). Note, this null model does not account for PCR-duplicates, and the assumption of a simple sampling distribution is probably simplistic.

Parameters: snp_array: the I x J x K 3-dimensional snp pileup array

FDR: Accepted false-discovery rate for a position.

error_rate: The expected rate of sequencing errors. For base-calling errors, the default is 1e-3, which corresponds to lowest Phred quality score considered when ConTExt builds SNP pileups (30).

depth_cutoff: A statistic will only be computed for a consensus position if the read depth at the position is higher than the cutoff in every sample.

Returns:

major_allele_prop: An I x K array indicating the major allele proportion at positon k in sample i

var_allele_freq: A length K array the fraction of samples containing a variant allele in at least one repeat unit

pos: The indices of consensus allele positions which survive the cutoff

6.3 Computing F_{st}

To compute F_{st} at each positions of a repeat:

rpt_fst=ComputeFst(rpt_array)

This computes F_{st} on allele proportion at each locus following Thomas Nagylaki "Fixation Indices in subdivided populations."

Parameters: snp_array: the I x J x K 3-dimensional snp pileup array pop: An array indicating sample labels. For the GDL, this stored as a global variable which is used by default.

Returns a length K array.

6.4 Principle Component Analysis

PCA can be carried out on the mean-centered major allele proportions with the following function:

pca_rpt= PlotPCA(rpt_prop)

This will plot scatterplots for the first 3 PCs and return an object storing the PCA.

A linear model can be used to interpret the PCA:

pca_rpt, lin_rpt, centered_prop= PlotPCA(rpt_prop, rpt_pos, lin_method={'EN'/'RR'})

There are three available methods for fitting the linear model which can be fed to the lin_method parameter:

EN: Elastic Net — Accomplishes feature selection and captures correlated variables. This is the preferred method for selecting the positions driving a PCA, but can be very slow (may take a few minutes)

RR: Ridge Regression — Does not select features, but does identify how strongly each feature drives the PCA. Very fast and recommend for exploring data.

RL: Randomized Lasso — Accomplishes feature select. Not particularly recommended. Elastic Net is better.

3-fold cross-validation determines the Elastic Net alpha and lambda parameters, and LOOCV determines Ridge Regression regularization parameter.

Biplots can be plotted with:

PlotBiplots(pca_rpt, lin_rpt)

6.5 Plotting the allele proportion distribution for the major allele at many positions

It is possible to automatically create plots of the allele proportion distribution for many populations. For example, to create a plot in the specified directory for each position with an Fst greater than .2:

PlotAllListedPositions(outDirectory, rpt_array,rpt_pos, (rpt_fst>=.2)):