TRTools

Gymreklab

Feb 21, 2024

CONTENTS

TRTools includes a variety of utilities for filtering, quality control and analysis of tandem repeats downstream of genotyping them from next-generation sequencing. It supports multiple recent genotyping tools (see below).

See full documentation and examples at https://trtools.readthedocs.io/en/stable/.

If you use TRTools in your work, please cite: Nima Mousavi, Jonathan Margoliash, Neha Pusarla, Shubham Saini, Richard Yanicky, Melissa Gymrek. (2020) TRTools: a toolkit for genome-wide analysis of tandem repeats. Bioinformatics. (https://doi.org/10.1093/bioinformatics/btaa736)

CHAPTER

ONE

INSTALL

Note: TRTools supports Python versions 3.8 and up. We do not officially support python version 3.7 as it is end of life, but we believe TRTools likely works with it from previous testing results.

1.1 With conda

```
conda install -c conda-forge -c bioconda trtools
```

Optionally install bcftools which is used to prepare input files for TRTools (and ART which is used by simTR) by running:

```
conda install -c conda-forge -c bioconda bcftools art
```

1.2 With pip

First install htslib (which contains tabix and bgzip). Optionally install bcftools. These are used to prepare input files for TRTools and aren't installed by pip.

Then run:

```
pip install --upgrade pip
pip install trtools
```

Note: TRTools installation may fail for pip version 10.0.1, hence the need to upgrade pip first

Note: if you will run or test simTR, you will also need to install ART. The simTR tests will only run if the executable art_illumina is found on your PATH. If it has been installed, which art_illumina should return a path.

1.3 From source

To install from source (only recommended for development) clone the TRTools repository from github and checkout the branch you're interested in:

```
git clone -b master https://github.com/gymrek-lab/TRTools
cd TRTools/
```

Now, create 1) a conda environment with our development tools and 2) a virtual environment with our dependencies and an editable install of TRTools:

```
conda env create -n trtools -f dev-env.yml
conda run -n trtools poetry install
```

Now, whenever you'd like to run/import pytest or TRTools, you will first need to activate both environments:

```
conda activate trtools
poetry shell
```

1.4 With Docker

Please refer to the biocontainers registry for TRTools for all of our images. To use the most recent release, run the following command:

```
docker pull quay.io/biocontainers/trtools:latest
```

CHAPTER

TWO

TOOLS

TRTools includes the following tools.

- mergeSTR: a tool to merge VCF files across multiple samples genotyped using the same tool
- dumpSTR: a tool for filtering VCF files with TR genotypes
- qcSTR: a tool for generating various quality control plots for a TR callset
- statSTR: a tool for computing various statistics on VCF files
- compareSTR: a tool for comparing TR callsets
- associaTR: a tool for testing TR length-phenotype associations (e.g., running a TR GWAS)
- prancSTR: a tool for identifying somatic mosacisim at TRs. Currently only compatible with HipSTR VCF files. (*beta mode*)
- simTR: a tool for simulating next-generation sequencing reads from TR regions. (*beta mode*)

Type `<command> --help` to see a full set of options.

It additionally includes a python library, `trtools`, which can be accessed from within Python scripts. e.g.:

```
import trtools.utils.utils as stls
allele_freqs = {5: 0.5, 6: 0.5} # 50% of alleles have 5 repeat copies, 50% have 6
stls.GetHeterozygosity(allele_freqs) # should return 0.5
```

CHAPTER

THREE

USAGE

We recommend new users start with the example commands described in the command-line interface for each tool. We also suggest going through our vignettes that walk through some example workflows using TRTools.

CHAPTER

FOUR

SUPPORTED TR CALLERS

TRTools supports VCFs from the following TR genotyping tools:

- AdVNTR
- ExpansionHunter
- GangSTR version 2.4 or higher
- HipSTR
- PopSTR version 2 or higher

See our description of the features and example use-cases of each of these tools.

CHAPTER

FIVE

TESTING

After you've installed TRTools, we recommend running our tests to confirm that TRTools works properly on your system. Just execute the following:

```
test_trtools.sh
```

CHAPTER SIX

DEVELOPMENT NOTES

- TRTools only currently supports diploid genotypes. Haploid calls, such as those on male chrX or chrY, are not yet supported but should be coming soon.

CHAPTER SEVEN

CONTACT US

Please submit an issue on the trtools github

CHAPTER

EIGHT

CONTRIBUTING

We appreciate contributions to TRTools. If you would like to contribute a fix or new feature, follow these guidelines:

1. Consider discussing your solution with us first so we can provide help or feedback if necessary.
2. Install TRTools from source *as above*.
3. Fork the TRTools repository.
4. Create a branch off of `master` titled with the name of your feature.
5. Make your changes.
6. If you need to add a dependency or update the version of a dependency, you can use the `poetry add` command.
   - You should specify a version constraint when adding a dependency. Use the oldest version compatible with your code. Don't worry if you're not sure at first, since you can (and should!) always update it later. For example, to specify a version of `numpy>=1.23.0`, you can run `poetry add 'numpy>=1.23.0'`.
   - Afterwards, double-check that the `poetry.lock` file contains 1.23.0 in it. **All of our dependencies should be locked to their minimum versions at all times.** To downgrade to a specific version of `numpy` in our lock file, you can explicitly add the version via `poetry add 'numpy==1.23.0'`, manually edit the pyproject.toml file to use a >= sign in front of the version number, and then run `poetry lock --no-update`.
7. Document your changes.
   - Ensure all functions, modules, classes etc. conform to numpy docstring standards.

     If applicable, update the REAMDEs in the directories of the files you changed with new usage information.
   - New doc pages for the website can be created under `<project-root>/doc` and linked to as appropriate.
   - If you have added significant amounts of documentation in any of these ways, build the documentation locally to ensure it looks good.

     `cd` to the `doc` directory and run `make clean && make html`, then view `doc/_build/html/index.html` and navigate from there
8. Add tests to test any new functionality. Add them to the `tests/` folder in the directory of the code you modified.
   - `cd` to the root of the project and run `poetry run pytest --cov=. --cov-report term-missing` to make sure that (1) all tests pass and (2) any code you have added is covered by tests. (Code coverage may **not** go down).
   - `cd` to the root of the project and run `nox` to make sure that the tests pass on all versions of python that we support.
9. Submit a pull request (PR) **to the master branch** of the central repository with a description of what changes you have made. Prefix the title of the PR according to the conventional commits spec. A member of the TRTools team

will reply and continue the contribution process from there, possibly asking for additional information/effort on your part.

- If you are reviewing a pull request, please double-check that the PR addresses each item in our PR checklist

CHAPTER

NINE

PUBLISHING

If you are a TRTools maintainer and wish to publish changes and distribute them to PyPI and bioconda, please see PUBLISHING.rst in the root of the git repo. If you are a community member and would like that to happen, contact us (see above).

CHAPTER

TEN

TABLE OF CONTENTS

10.1 TRTools Vignettes

The following vignettes show example usecases of TRTools.

10.1.1 Computing per-locus TR statistics

Tools used: mergeSTR, statSTR

This vignette shows how to use `mergeSTR` to merge two VCF files and `statSTR` to compute statistics across different sample groups for an example TR locus. It uses the example VCF files `ceu_ex.vcf.gz` and `yri_ex.vcf.gz` available at https://github.com/gymrek-lab/TRTools/tree/master/example-files. These VCFs were generated by GangSTR on samples sequenced by the 1000 Genomes Project. They have already been sorted and indexed.

After downloading the VCF files, we can use `mergeSTR` to merge them into a single VCF:

```
mergeSTR --vcfs ceu_ex.vcf.gz,yri_ex.vcf.gz --out merged
```

This will create the output `merged.vcf`. We should zip and index the file:

```
bgzip merged.vcf
tabix -p vcf merged.vcf.gz
```

Now, we can use `statSTR` to compute various TR-level statistics across all samples or different groups of samples. Below, we'll use statSTR to compute mean allele length, heterozygosity, and allele counts, separately for each population at each locus:

```
# Get the CEU and YRI sample lists
bcftools query -l yri_ex.vcf.gz > yri_samples.txt
bcftools query -l ceu_ex.vcf.gz > ceu_samples.txt

# Run statSTR on region chr21:35348646-35348646 (hg38)
statSTR \
--vcf merged.vcf.gz \
--samples yri_samples.txt,ceu_samples.txt \
--sample-prefixes YRI,CEU \
--out stdout \
--mean --het --acount \
--use-length \
--region chr21:34351482-34363028
```

Let's go through what each option did:

- `--vcf` gives the required input VCF file. Since we'll be using the `--region` option, the file needs to be sorted, bgzipped, and indexed.
- `--out` is the required output prefix. Use `stdout` as in the above example to output to the terminal screen.
- `--samples` gives a comma-separated list of files containing sample groups to plot separately. Here we want to compute YRI and CEU stats separately and have provided two files with the teo sample lists.
- `--sample-prefixes` tells the script what to name the outputs for the different sample groups. These labels get used in the column headers of the output file.
- `--region`: tells the script which region to process. Here we're just processing a couple loci for the sake of example.
- `--mean`: outputs a column with the mean repeat length (in terms of number of repeat units)
- `--het`: outputs a column with heterozygosity of the locus
- `--acount`: outputs a column with allele counts at each locus
- `--use-length`: Means to consider only allele lengths, rather than sequences, when computing statistics.

Here, since we have two labeled sample lists, columns will be named: `<stat>-YRI` and `<stat>-CEU`, where stat is either mean, het, or acount.

The above command outputs the following:

```
chrom   start   end     acount-YRI      acount-CEU      het-YRI het-CEU mean-YRI       ␣
↪mean-CEU
chr21   34351482        34351499        9.0:190,10.0:24 9.0:192,10.0:4  0.
↪19914403004629233      0.03998334027488548     9.11214953271028        9.020408163265305
chr21   34358397        34358411        14.0:19,15.0:180,16.0:15        15.0:196       ␣
↪0.279718752729496       0.0     14.981308411214952      15.0
chr21   34361277        34361288        3.0:214 3.0:196 0.0     0.0     3.0     3.0
chr21   34362436        34362447        3.0:214 3.0:196 0.0     0.0     3.0     3.0
chr21   34362864        34362881        3.0:212 3.0:196 0.0     0.0     3.0     3.0
chr21   34363028        34363039        4.0:65,5.0:149  4.0:142,5.0:54  0.
↪42296270416630277      0.3992086630570595      4.696261682242991       4.275510204081632
```

So we can see, for example, that at the TR at chr21:34351482-34351499, there were 109 YRI alleles with 9 repeats and 24 with 10 repeats. Similarly, the mean length was 9.1 in YRI and 9.0 in CEU.

10.1.2 Plotting allele length distributions by population group

Tools used: mergeSTR, statSTR

This vignette shows how to use `mergeSTR` to merge two VCF files and `statSTR` to plot allele frequencies across different sample groups for an example TR locus. It uses the example VCF files `ceu_ex.vcf.gz` and `yri_ex.vcf.gz` available at https://github.com/gymrek-lab/TRTools/tree/master/example-files. These VCFs were generated by GangSTR on samples sequenced by the 1000 Genomes Project. They have already been sorted and indexed.

After downloading the VCF files, we can use `mergeSTR` to merge them into a single VCF:

```
mergeSTR --vcfs ceu_ex.vcf.gz,yri_ex.vcf.gz --out merged
```

This will create the output `merged.vcf`. We should zip and index the file:

```
bgzip merged.vcf
tabix -p vcf merged.vcf.gz
```

Now, we can use `statSTR` to compute or plot statistics for one or more TR loci across all samples or one or more subsets of samples in the file. Below, we show an example plotting the distribution of allele lengths in CEU and YRI populations separately for a single TR:

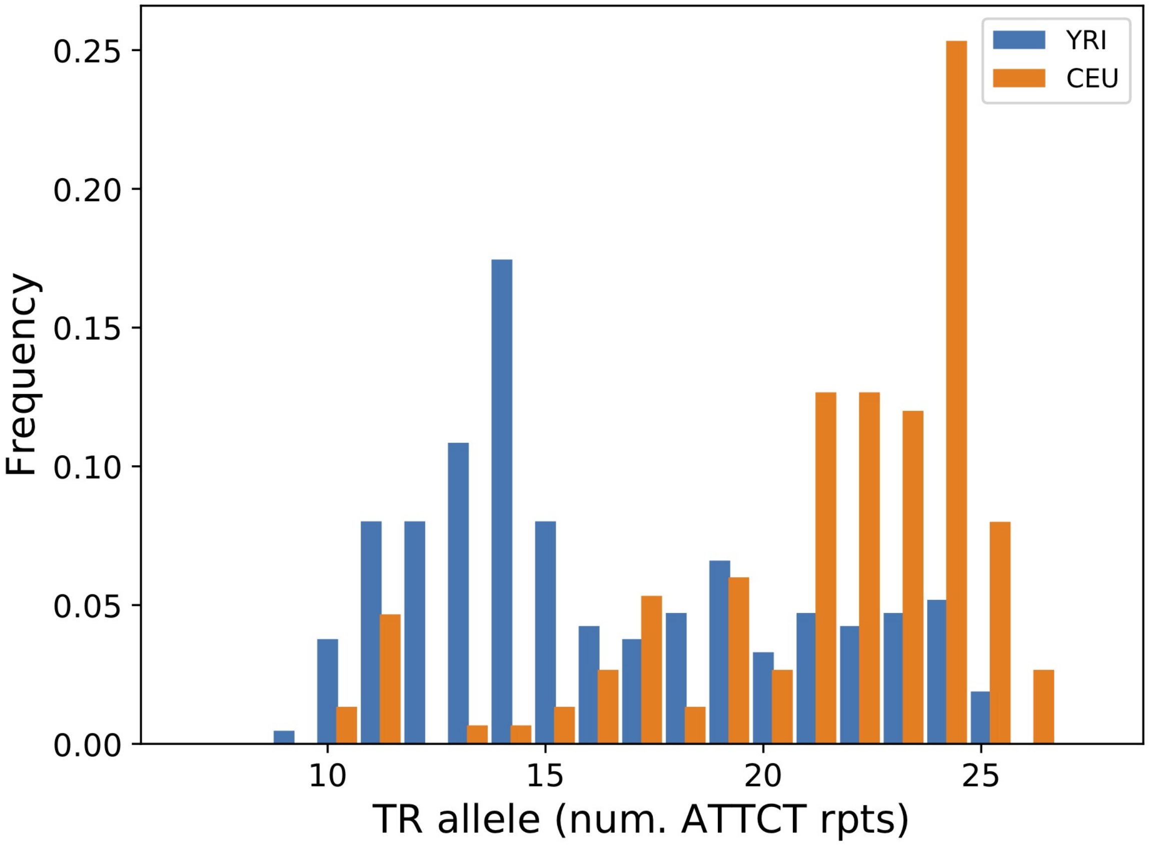
```
# Get the CEU and YRI sample lists
bcftools query -l yri_ex.vcf.gz > yri_samples.txt
bcftools query -l ceu_ex.vcf.gz > ceu_samples.txt

# Run statSTR on region chr21:35348646-35348646 (hg38)
statSTR \
--vcf merged.vcf.gz \
--samples yri_samples.txt,ceu_samples.txt \
--sample-prefixes YRI,CEU \
--region chr21:35348646-35348646 \
--out yri_ceu_runx1 \
--plot-afreq
```

Let's go through what each option did:

- `--vcf` gives the required input VCF file. Since we'll be using the `--region` option, the file needs to be sorted, bgzipped, and indexed.
- `--out` is the required output prefix.
- `--samples` gives a comma-separated list of files containing sample groups to plot separately. Here we want to plot YRI and CEU samples separately and have provided two files with the teo sample lists.
- `--sample-prefixes` tells the script what to name the outputs for the different sample groups. These labels get used in the legend on the plot.
- `--region`: tells the script which region to process. Here we're just processing a single TR locus.
- `--plot-afreq`: tells the script to output a plot with the allele frequencies at this locus. Note the script will only output up to 10 plots in a single run. Plotting is meant to be run on a small set of loci.

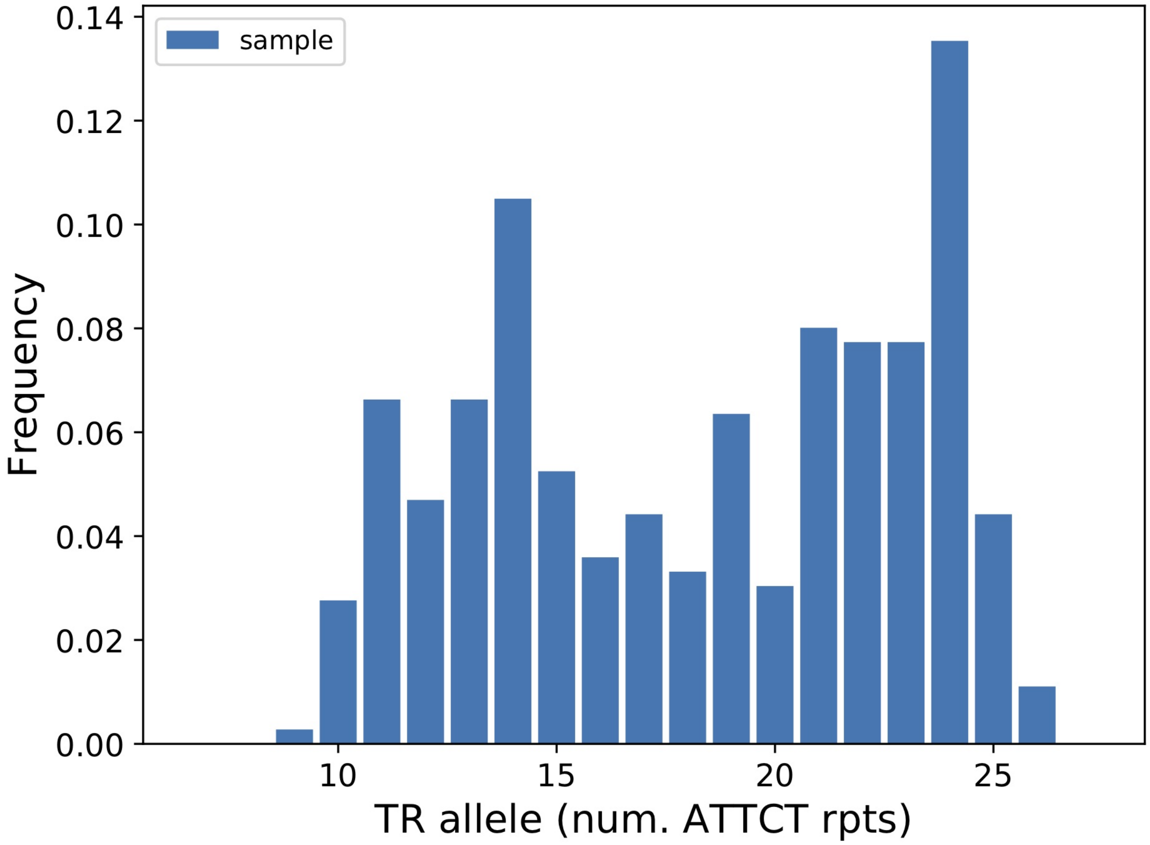
This should produce the output file `yri_ceu_runx1-chr21-35348646.pdf`, which is shown below.



You could have also run the command without specifying sample lists, which will plot all samples together:

```
statSTR \
--vcf merged.vcf.gz \
--region chr21:35348646-35348646 \
--out all_runx1 \
--plot-afreq
```

which outputs the plot below:



10.1.3 Comparing TR calls across different parameter sets

Tools used: compareSTR

This vignette shows how to use `compareSTR` to compare two VCF files generated using the same set of reference TRs. In this example, we use VCF files `c57_ex1.vcf.gz` and `c57_ex2.vcf.gz` available at https://github.com/gymrek-lab/TRTools/tree/master/example-files. These VCF files were generated by GangSTR on a mouse dataset using two different sets of stutter parameters.

To run `compareSTR`:

```
compareSTR \
--vcf1 c57_ex1.vcf.gz \
--vcf2 c57_ex2.vcf.gz \
--vcftype1 gangstr \
--vcftype2 gangstr \
--out c57-compare \
--stratify-fields DP \
--stratify-binsizes 0:50:10 \
--bubble-min -10 --bubble-max 10
```

Let's go through what each option did:

- `--vcf1` and `--vcf2` give the name of the two VCF files being compared.
- `--vcftype1` and `--vcftype2` give the types of the two VCF files being compared

- `--out` is the required output prefix.
- `--stratify-fields DP` and `--stratify-binsizes 0:50:10` tell the script to compute overall concordance metrics stratifying by the DP field (coverage) in bins of 10 ranging from 0 to 50. Since we didn't specify which file to apply the stratification to with `--stratify-file`, it gets applied to both.
- `--bubble-min -10 --bubble-max 10` give the axis range of the bubble plot to show (see below).

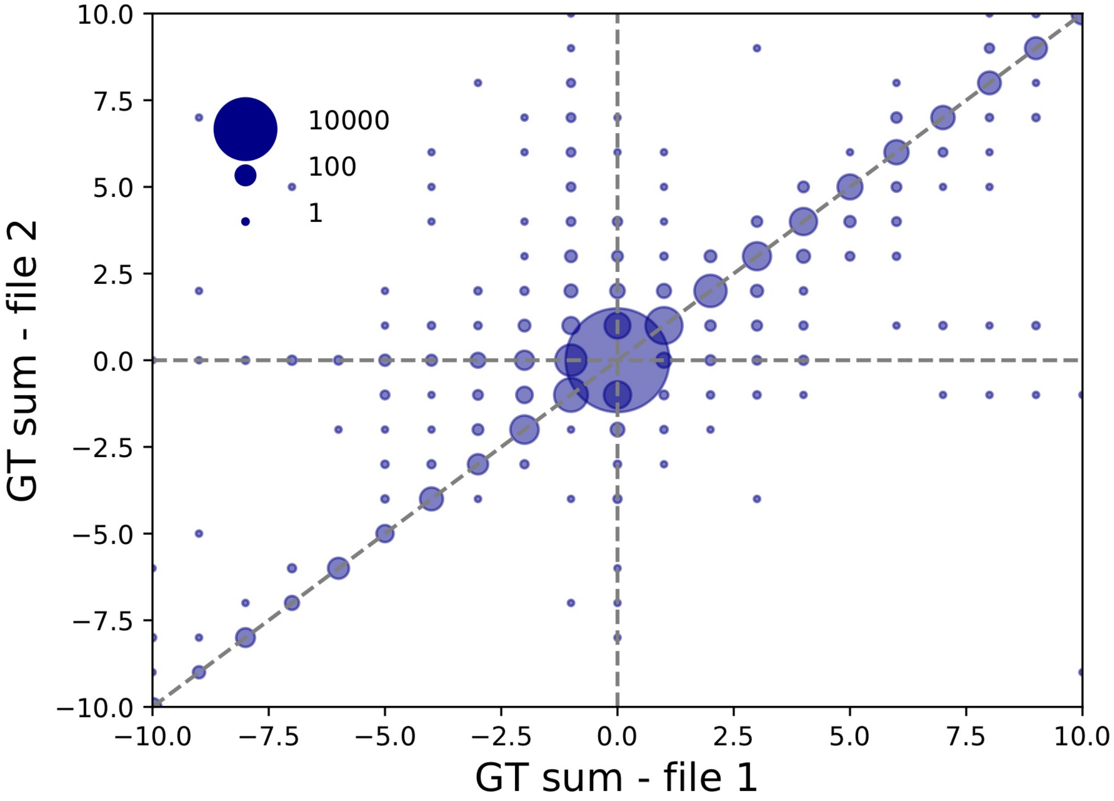
This will output a number of files. We'll peek at a couple of them.

- `c57-compare-overall.tab` will give overall concordance info:

```
period  DP      concordance-seq concordance-len r2      numcalls
ALL     NA      0.9760415527610716      0.9760415527610716      0.9909303005952199 ␣
↪    91450
ALL     0.0-10.0        1.0     1.0     1.0     266
ALL     10.0-20.0       0.9426900584795321      0.9426900584795321      0.
↪9995028621738471       855
ALL     20.0-30.0       0.9285714285714286      0.9285714285714286      0.
↪9950024662641009       1218
ALL     30.0-40.0       0.9548577036310107      0.9548577036310107      0.
↪9780821396586636       3057
ALL     40.0-50.0       0.9739065606361829      0.9739065606361829      0.
↪9880478071563313       8048
```

Here, the first line gives the overall concordance. The lines below are stratified by DP value. For each set, the concordance (percent of calls matching), r2 (Pearson correlation between allele calls), and number of calls in each group is shown.

- `c57-compare-callcompare.tab` gives a call by call comparison, which is useful for looking at exactly which loci/sample calls were discordant.
- `c57-compare-locuscompare.tab` and `c57-compare-samplecompare.tab` give locus and sample level concordance. In this case these are not very interesting since we have only one sample being compared. But in other settings, these files can be used to identify poorly performing samples or loci.
- `c57-compare-bubble-periodALL.pdf` plots the genotypes in file 1 vs. file 2. Genotypes are given in terms of the sum across both alleles of the repeat units different from the reference. Bubble sizes give the number of calls represented by each point. This plot is shown below:



Since we used `--bubble-min -10 --bubble-max 10`, the plot goes only from -10 to +10. This plot can be useful in evaluating the effects of varying different parameters on calling. For example, we can see from this plot that there are quite a few calls that are called longer in file 2 vs. file 1.

10.1.4 Comparing TR calls across different genotypers

Tools used: mergeSTR, compareSTR

This vignette shows how to use `mergeSTR` to merge VCFs from multiple samples into a single VCF, and `compareSTR` to compare VCF files generated by different genotypers (HipSTR and ExpansionHunter) using the same set of reference TRs. In this example, we use VCF files available at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

- `NA12878_chr21_eh.sorted.vcf.gz`, `NA12891_chr21_eh.sorted.vcf.gz`, and `NA12892_chr21_eh.sorted.vcf.gz` generated using ExpansionHunter on three separate samples
- `trio_chr21_hipstr.sorted.vcf.gz` generated using HipSTR run jointly on all three samples.

Both VCFs were generated using the same reference set of TRs to make them comparable. We'll also need the file `hg19.fa.fai` for adding the appropriate header lines to the ExpansionHunter VCF files. (You can generate this file from a reference genome fasta file using `samtools faidx`).

First, we'll want to merge the three samples called separately by ExpansionHunter into a single VCF file using `mergeSTR`. We'll need to do one VCF cleanup using *bcftools reheader* to make sure the appropriate contig lines are present in the VCF file:

```
# Add contig lines to the headers
bcftools reheader -f hg19.fa.fai -o NA12878_chr21_eh.reheader.vcf.gz  NA12878_chr21_eh.
↪sorted.vcf.gz
bcftools reheader -f hg19.fa.fai -o NA12891_chr21_eh.reheader.vcf.gz  NA12891_chr21_eh.
↪sorted.vcf.gz
bcftools reheader -f hg19.fa.fai -o NA12892_chr21_eh.reheader.vcf.gz  NA12892_chr21_eh.
↪sorted.vcf.gz
tabix -p vcf NA12878_chr21_eh.reheader.vcf.gz
tabix -p vcf NA12891_chr21_eh.reheader.vcf.gz
tabix -p vcf NA12892_chr21_eh.reheader.vcf.gz

# Merge all the EH samples to one VCF
mergeSTR --vcfs NA12878_chr21_eh.reheader.vcf.gz,NA12891_chr21_eh.reheader.vcf.gz,
↪NA12892_chr21_eh.reheader.vcf.gz \
    --out trio_chr21_eh

# Bgzip and index the output VCF to get ready for compareSTR
bgzip trio_chr21_eh.vcf
tabix -p vcf trio_chr21_eh.vcf.gz
```

Now, we have a file from each tool (`trio_chr21_hipstr.sorted.vcf.gz` and `trio_chr21_eh.vcf.gz`) containing genotypes for all three samples. We can use `compareSTR` to compare these:

```
compareSTR --vcf1 trio_chr21_hipstr.sorted.vcf.gz --vcf2 trio_chr21_eh.vcf.gz \
   --vcftype1 hipstr --vcftype2 eh --out hipstr_vs_eh
```

This command tells `compareSTR` to compare these two VCFs. By specifying `--vcftype1` and `--vcftype2` we are telling it what format the two VCFs are in. Note, TRTools will try to automatically infer this anyway, and make sure what we infer matches what was specified.

This should output several text files:

- `hipstr_vs_eh-overall.tab` will give overall concordance info:

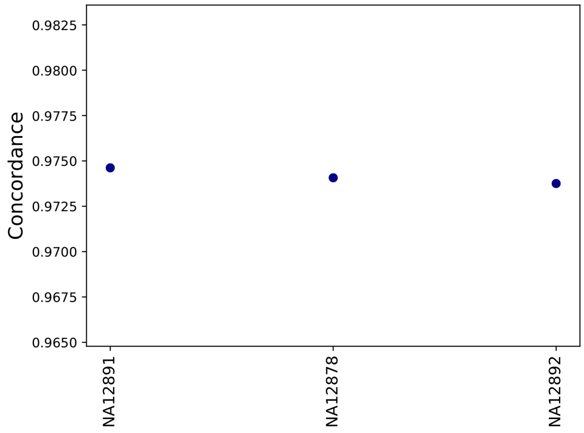
  ```
  period     concordance-seq    concordance-len    r2                 numcalls
  ALL        0.9533784988411481 0.9741486896059903 0.734005622224775 11218
  ```

Since we did not specify the `--period` option, we are considering "ALL" repeat units. This file shows the overall percent of matching calls based on allele sequences (`concordance-seq`) and allele lengths (`concordance-len`), the squared Pearson between genotypes (sum of allele lengths) (`r2`), and the number of calls compared (`numcalls`).

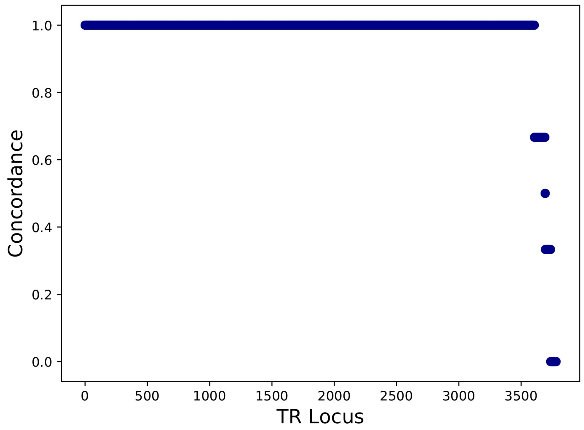
- `hipstr_vs_eh-callcompare.tab` gives a call by call comparison.
- `hipstr_vs_eh-locuscompare.tab` gives per-locus comparison statistics.
- **`hipstr_vs_eh-samplecompare.tab` gives per-sample comparison statistics::** sample metric-conc-seq metric-conc-len numcalls NA12891 0.9540475554368154 0.9746192893401016 3743 NA12878 0.9526864474739375 0.9740711039828923 3741 NA12892 0.953401178361007 0.9737546866630958 3734

`mergeSTR` will also output several plots to visualize the comparison results:

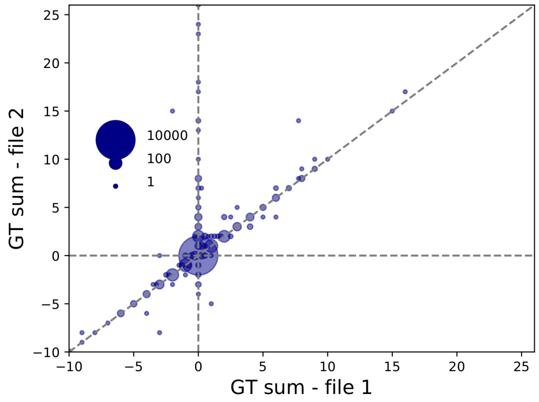
- `hipstr_vs_eh-samplecompare.pdf`



- `hipstr_vs_eh-locuscompare.pdf`



- `hipstr_vs_eh-bubble-periodALL.pdf`



10.1.5 Filtering and QC of VCFs

Tools used: dumpSTR, qcSTR

This vignette shows how to use `dumpSTR` to filter a VCF and `qcSTR` to visualize some basic QC metrics. For this example, we use the file `trio_chr21_popstr.sorted.vcf.gz` available at https://github.com/gymrek-lab/TRTools/tree/master/example-files. This file was generated on samples NA12878, NA12891, and NA12892 using popSTR.

First, let's perform some filtering on the VCF:

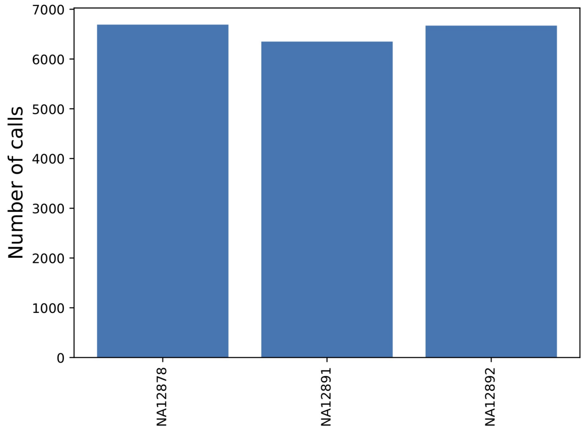
```
dumpSTR --vcf trio_chr21_popstr.sorted.vcf.gz --popstr-require-support 2 --popstr-min-
↪call-DP 10 \
    --out popstr-filtered --min-locus-callrate 1
bgzip -f popstr-filtered.vcf
tabix -p vcf popstr-filtered.vcf.gz
```

This command filters calls with depth of less than 10 or with alleles supported by less than 2 reads, and loci with any missing genotypes. Now, we can run `qcSTR` on the filtered VCF:

```
qcSTR --vcf popstr-filtered.vcf.gz --out popstr-qc
```

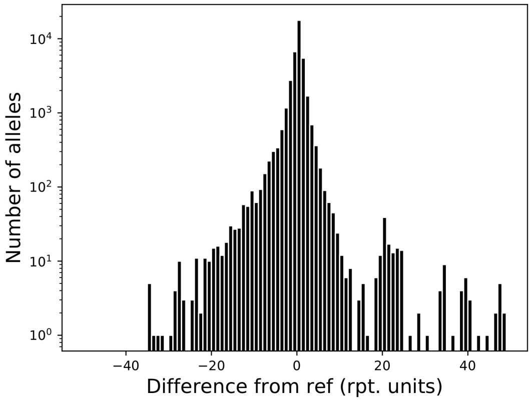
This will output the following files:

- `popstr-qc-sample-callnum.pdf`



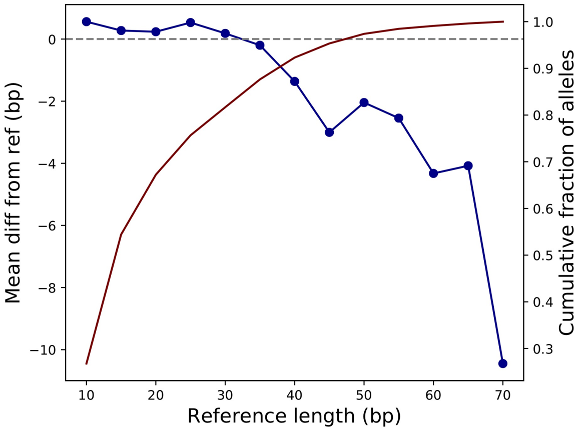
This shows the number of calls per sample.

- `popstr-qc-diffref-histogram.pdf`



This shows the distribution of allele sizes relative to the reference genome.

- `popstr-qc-diffref-bias.pdf`



This shows the difference in allele size from the reference as a function of the reference length. We can see as expected calls are biased toward deletions for longer TRs.

10.2 Command-Line Tools

TRTools offers a variety of command-line tools for performing manipulations to TR genotype VCFs. (These tools have ‘STR’ in their name, but are applicable to all TR VCFs).

10.2.1 DumpSTR

DumpSTR filters VCF files with TR genotypes, performing call-level and locus-level filtering, and outputs a filtered VCF file.

Usage

To run dumpSTR use the following command:

```
dumpSTR \
  --vcf <vcf file> \
  --out <string> \
  [filter options]
```

Required parameters:

- `--vcf <VCF>` VCF file to filter that has been generated by a supported genotyping tool.
- `--out <string>` prefix to name output files.

DumpSTR will output a filtered VCF file named `$out.vcf`, a sample log file `$out.samplog.tab`, and a locus log file `$out.loclog.tab`. See a description of output files below.

Other general parameters:

- `--vcftype <string>`: Which genotyping tool generated the input VCF. Default = `auto`. Necessary if it cannot be automatically inferred. One of: `gangstr`, `advntr`, `hipstr`, `eh`, `popstr`.
- `--zip` to output a bgzipped filtered VCF (`$out.vcf.gz`) and tabix index (`$out.vcf.gz.tbi`) instead of `$out.vcf`.
- `--num-records <int>`: only process this many records from the input VCF file

The available filters are described below.

See *Example Commands* for running dumpSTR on different supported TR genotypers using example VCFs in this repository.

Filter options

DumpSTR offers the following types of filters:

Locus-level filters

These filters are not specific to any tool and can be applied to any VCF file:

- `--min-locus-callrate <float>`: Filters loci with too few calls
- `--min-locus-hwep <float>`: Filters loci departing from Hardy Weinberg equilibrium at some p-value threshold. Based on a two-sided binomial test comparing the observed vs. expected percent of calls that are homozygous
- `--min-locus-het <float>`: Filters loci with low heteroyzgosity. Heterozygosity is equal to:

$$1 - \sum_i p_i^2$$

where p_i is the frequency of allele i

- `--max-locus-het <float>`: Filters loci with high heterozygosity
- `--use-length`: Use allele lengths, rather than sequences, to compute heterozygosity and HWE (only relevant for HipSTR, which reports sequence level differences in TR alleles)

- `--filter-regions <BEDFILE[,BEDFILE12,...]>`: Filter TRs overlapping the specified set of regions. Must be used with `--filter-regions-names`. Can supply a comma-separated list to each to apply multiple region filters. Bed files must be sorted and tabix-indexed. Note that regions are 0-based and inclusive of the start position, but exclusive of the end position.
- `--filter-regions-names <string[,string2,...]>`: Filter names for each BED file specified in `--filter-regions`.
- `--filter-hrun`: Filter repeats with long homopolymer runs. Only used for HipSTR VCF files otherwise ignored. Filters pentanucleotides with homopolymer runs of 5bp or longer, or hexanucleotides with homopolymer runs of 6bp or longer.
- `--drop-filtered`: Do not output loci that were filtered, or loci with no calls remaining after filtering.

TRs passing all locus-level filters will be marked as "PASS" in the FILTER field. Those failing will have a list of failing filters in the FILTER field. If a TR has no calls remaining after applying the call-level filters, then it will be marked as "NO_CALLS_REMAINING" and not "PASS". If `drop-filtered` is specified, only loci passing all filters will be output. Otherwise failing loci will be output as well. In that case the only indication that they have failed will be the FILTER field. The call-level genotypes will still be present in the output VCF.

Call-level filters

Different call-level filters are available for each supported TR genotyping tool. Genotypes which fail any call-level filters are replaced by no calls and the FILTER format field for that call is set to the reason(s) that call was filtered. For example, if running with `--hipstr-min-call-DP 100`, a call with DP 34 would be filtered with the text `HipSTRCallMinDepth100_34`. Calls which were already nocalls before the dumpSTR run are left unchanged and their FILTER format field is set to NOCALL.

Caveat: No call level filters have yet been written for VCFs imputed by Beagle.

AdVNTR call-level filters

- `--advntr-min-call-DP <int>`: Minimum call coverage. Based on DP field.
- `--advntr-max-call-DP <int>`: Maximum call coverage. Based on DP field.
- `--advntr-min-spanning <int>`: Minimum spanning read count (SR field)
- `--advntr-min-flanking <int>`: Minimum flanking read count (FR field)
- `--advntr-min-ML <float>`: Minimum value of maximum likelihood (ML field)

ExpansionHunter call-level filters

- `--eh-min-call-LC <int>`: Minimum call coverage. Based on LC field.
- `--eh-max-call-LC <int>`: Maximum call coverage. Based on LC field.

GangSTR call-level filters

- `--gangstr-min-call-DP <int>`: Minimum call coverage. Based on DP field.
- `--gangstr-max-call-DP <int>`: Maximum call coverage. Based on DP field.
- `--gangstr-min-call-Q <float>`: Minimum call quality score. Based on Q field.
- `--gangstr-expansion-prob-het <float>`: Expansion prob-value threshold. Filters calls with probability of heterozygous expansion less than this. Based on QEXP field.
- `--gangstr-expansion-prob-hom <float>`: Expansion prob-value threshold. Filters calls with probability of homozygous expansion less than this. Based on QEXP field.
- `--gangstr-expansion-prob-total <float>`: Expansion prob-value threshold. Filters calls with probability of homozygous or heterozygous expansion less than this. Based on QEXP field.
- `--gangstr-filter-span-only`: Filter out all calls that only have spanning read support. Based on RC field.
- `--gangstr-filter-spanbound-only`: Filter out all reads except spanning and bounding. Based on RC field.
- `--gangstr-filter-badCI`: Filter regions where the ML estimate is not in the CI. Based on REPCN and REPCI fields.

HipSTR call-level filters

- `--hipstr-max-call-flank-indel <float>`: Maximum call flank indel rate. Computed as DFLANKINDEL/DP
- `--hipstr-max-call-stutter <float>`: Maximum call stutter rate. PCR stutter artifacts add or remove copies of an STR's motif to sequencing reads, resulting in observed STR sizes that differ from the size of the underlying genotype. (Source). Computed as DSTUTTER/DP
- `--hipstr-min-supp-reads <int>`: Minimum supporting reads for each allele. Based on ALLREADS and GB fields
- `--hipstr-min-call-DP <int>`: Minimum call coverage. Based on DP field.
- `--hipstr-max-call-DP <int>`: Maximum call coverage. Based on DP field.
- `--hipstr-min-call-Q <float>`: Minimum call quality score. Based on Q field.

PopSTR call-level filters

- `--popstr-min-call-DP <int>`: Minimum call coverage. Based on DP field.
- `--popstr-max-call-DP <int>`: Maximum call coverage. Based on DP field.
- `--popstr-require-support <int>`: Require each allele call to have at least n supporting reads. Based on AD field.

Output files

DumpSTR outputs the following files:

- `$out.vcf`: Filtered VCF file. Filtered loci have a list of failing filters in the FILTER column. An additional FORMAT:FILTER field is added to each call. This is set to PASS for passing calls. For failing calls, this is set to a list of filter reasons and the genotype is set to missing.
- `$out.samplog.tab`: Output sample-level log info. This is a tab-delimited file with columns: sample, number of calls, and mean coverage at that sample across calls that survived dumpSTR filtering. This file also contains a column for each call-level filter indicating how many calls for that sample were filtered due to that reason. e.g. column "AdVNTRCallMinDepth" would indicate the number of adVNTR calls for that sample filtered due to low call depth (based on `--advntr-min-call-DP`). Some calls are filtered for more than one reason, so the sum of filtered calls across all reasons will likely be more than the number of filtered calls.
- `$out.loclog.tab`: Output locus-level log info. It contains the mean call rate at passing TR loci. It also contains a separate line for each filter with the number of TR loci failing that filter.

Example Commands

Below are `dumpSTR` examples using VCFs from supported TR genotypers. Data files can be found at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

```
# AdVNTR
dumpSTR --vcf NA12878_chr21_advntr.sorted.vcf.gz --advntr-min-call-DP 100 --out test_
↪dumpstr_advntr

# ExpansionHunter
dumpSTR --vcf NA12878_chr21_eh.sorted.vcf.gz --out test_dumpstr_eh --eh-min-call-LC 50 --
↪num-records 10 --drop-filtered

# GangSTR
dumpSTR --vcf trio_chr21_gangstr.sorted.vcf.gz --out test_dumpstr_gangstr --min-locus-
↪callrate 0.9 --num-records 10

# HipSTR
dumpSTR --vcf trio_chr21_hipstr.sorted.vcf.gz --vcftype hipstr --out test_dumpstr_hipstr␣
↪--filter-hrun --num-records 10

# PopSTR
dumpSTR --vcf trio_chr21_popstr.sorted.vcf.gz --out test_dumpstr_popstr --min-locus-
↪callrate 0.9 --popstr-min-call-DP 10 --num-records 100
```

10.2.2 StatSTR

StatSTR takes in a TR genotyping VCF file and outputs per-locus statistics.

Usage

To run statSTR use the following command:

```
statSTR
   --vcf <vcf file> \
   --out <prefix> \
   --thresh \
   [additional options]
```

Required parameters:

- `--vcf <string>`: The input TR VCF file
- `--out <string>`: The prefix to name output files. Set to stdout to write to standard output.

Optional general parameters:

- `--vcftype <string>`: The type of VCF file being processed. Default = `auto` Must be one of: `gangstr`, `advntr`, `hipstr`, `eh`, `popstr`.
- `--samples <string>`: A file containing a list of samples to include in computing statistics. If not given, all samples are used. To compute statistics for multiple groups of samples, you can give a comma-separated list of samples files. Sample files should list one sample per line, no header line. Samples not found in the VCF are silently ignored.
- `--sample-prefixes <string>`: The prefixes to name output for each samples group. By default uses 1, 2, 3 etc. Must be sample length as `--samples`.
- `--region <string>`: Restrict to specific regions (chrom:start-end). Requires the input VCF to be bgzipped and tabix indexed.
- `--precision <int>`: How much precision to use when writing stats (default = 3)

For specific statistics available, see below.

StatSTR outputs a tab-delimited file `<outprefix>.tab` with per locus statistics. See a description of the output file below.

See *Example Commands* below for example statSTR commands for different supported TR genotypers based on example data files in this repository.

Statistics options

The following options can be specified to compute various per-locus statistics. The column produced in the output file has the same name as the specified option:

- `--thresh`: Output the maximum observed allele length.
- `--afreq`: Output allele frequences. Comma-separated list of allele:freq. Only called alleles are included in the list. If there are no called alleles, '.' is emitted.
- `--acount`: Output allele counts. Comma-separated list of allele:count Only called alleles are included in the list. If there are no called alleles, '.' is emitted.
- `--nalleles`: Output the number of alleles per locus with at least a given frequency.
- `--nalleles-thresh`: Threshold for the frequency cutoff for `--nalleles`. Defaults to `1%` if not specified.
- `--hwep`: Output Hardy Weinberg p-values per locus.
- `--het`: Output heterozygosity of each locus.

- `--entropy`: Output the bit-entropy of the distribution of alleles at each locus. Entropy is a measurement of how hard it is to predict genotypes at a locus, where higher entropy values indicate more complex loci. See wikipedia for the mathematical definition of entropy.
- `--mean`: Output mean allele length.
- `--mode`: Output mode allele length.
- `--var`: Output variance of allele length.
- `--numcalled`: Output number of called samples.

The statistics `thresh, hewp, het, entropy, mean, mode, var` will output `nan` if there are no called alleles. The statistics `afreq, acount` will output `.` if there are no called alleles. The statistic `nalleles` will output `0` if there are no called alleles.

For genotypers which output allele sequences, `--use-length` will collapse alleles by length. (This is implicit for genotypers which only output allele lengths.)

Output file

StatSTR outputs a tab-delimited file with columns `chrom`, `start` and `end` plus an additional column for each statistic specified. If multiple sample groups are specified, instead there is one additional column for each group-by-statistic pair

Example Commands

Below are `statSTR` examples using VCFs from supported TR genotypers. Data files can be found at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

```
# AdVNTR
statSTR --vcf NA12878_chr21_advntr.sorted.vcf.gz \
      --out stdout \
      --afreq

# ExpansionHunter
statSTR --vcf NA12891_chr21_eh.sorted.vcf.gz \
      --out stats_eh \
      --numcalled

# GangSTR
statSTR --vcf trio_chr21_gangstr.sorted.vcf.gz \
       --out stats_gangstr \
       --numcalled \
       --mean

# HipSTR
statSTR --vcf trio_chr21_hipstr.sorted.vcf.gz \
      --vcftype hipstr \
      --out stats_hipstr \
      --acount \
      --afreq \
      --mean

# PopSTR
```

(continues on next page)

(continued from previous page)

```
statSTR --vcf trio_chr21_popstr.sorted.vcf.gz \
      --out stats_popstr \
      --mean \
      --samples ex-samples.txt
```

10.2.3 qcSTR

qcSTR generates plots that are useful for diagnosing issues in TR calling.

Usage

qcSTR takes as input a VCF file and outputs several plots in pdf format. To run qcSTR, use the following command:

```
qcSTR \
    --vcf <file.vcf> \
    --out <string> \
    [additional options]
```

Required Parameters:

- `--vcf <string>`: Input VCF file
- `--out <string>`: Prefix to name output files

General Optional Parameters:

- `--vcftype <string>`: Type of the input VCF file. Causes qcSTR to fail out if the file is any other type of VCF. Default = `auto`. Must be one of: `gangstr`, `advntr`, `hipstr`, `eh`, `popstr`.
- `--samples <string>`: File containing list of samples to include. If not specified, all samples are used. Samples in the list that are not included in the input vcf or are misspelled are silently ignored.
- `--period <int>`: Restrict to TRs with this motif length. e.g. to restrict to dinucleotide repeats, use `--period 2`.
- `--version`: Show the version number of qcSTR and exit.

If you wish to run qcSTR on a more complicated subset of the input VCF, we suggest you use `dumpSTR` or bcftools view to filter the input VCF first, and then run qcSTR on the vcf those commands outputed.

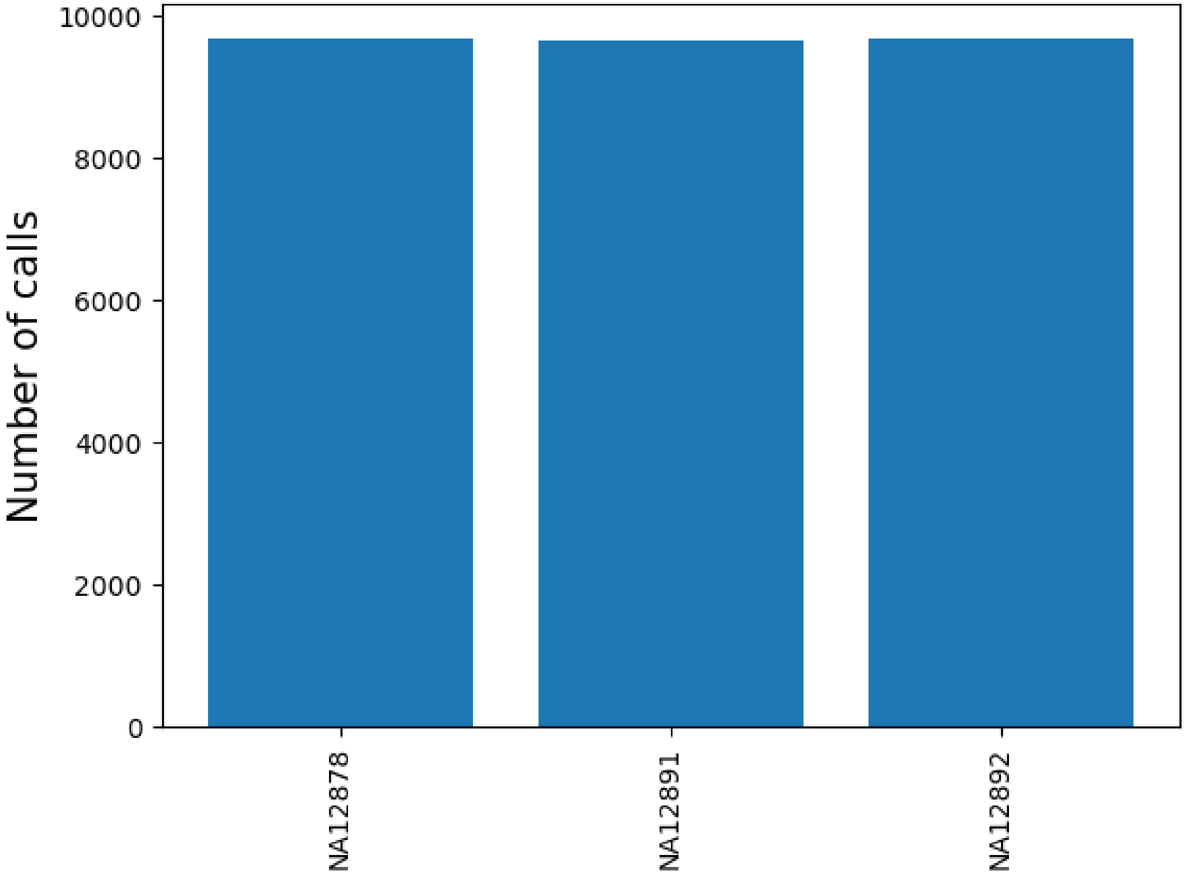
Additional options to customize various plots output by qcSTR are described in the sections *Quality Plot Options* and *Reference Bias Plot Options* below.

See *Example Commands* below for example qcSTR commands for different supported TR genotypers based on example data files in this repository.

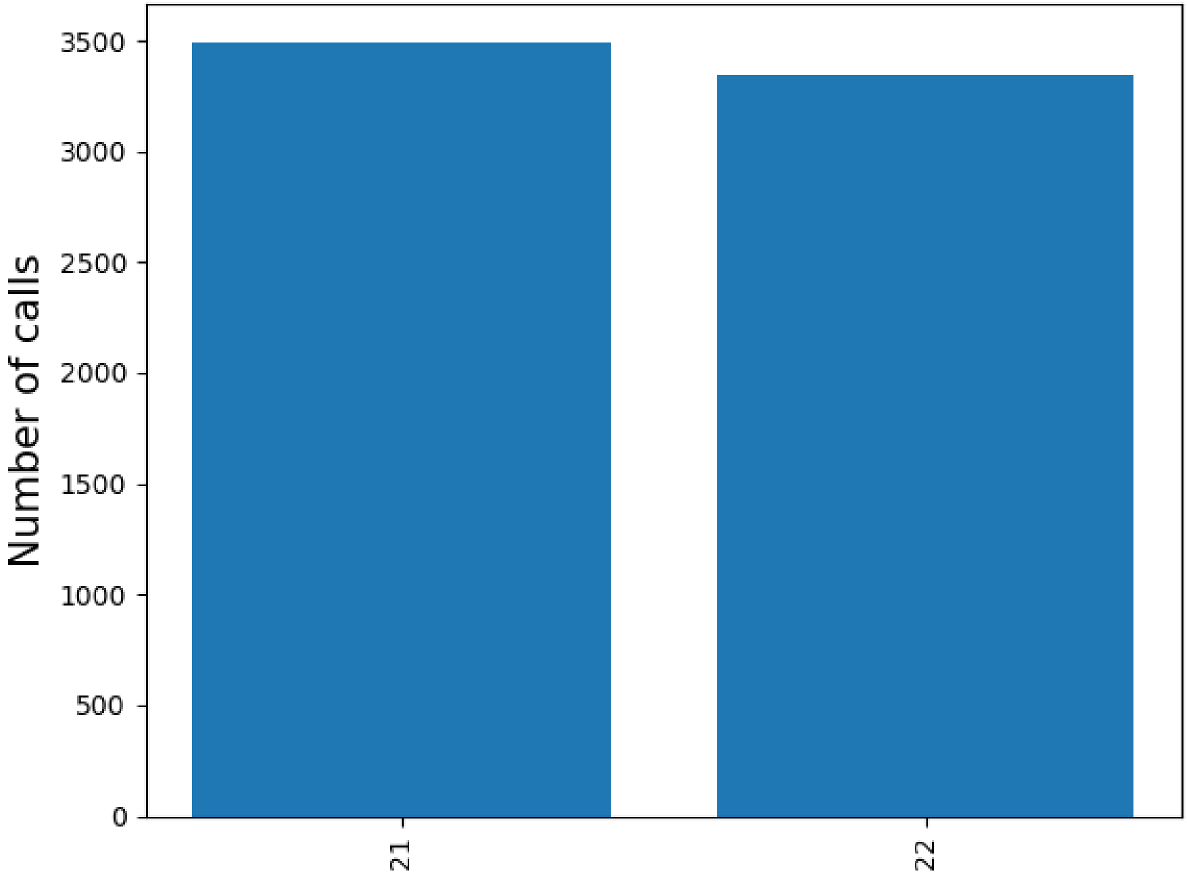
Outputs

qcSTR outputs the following plots:

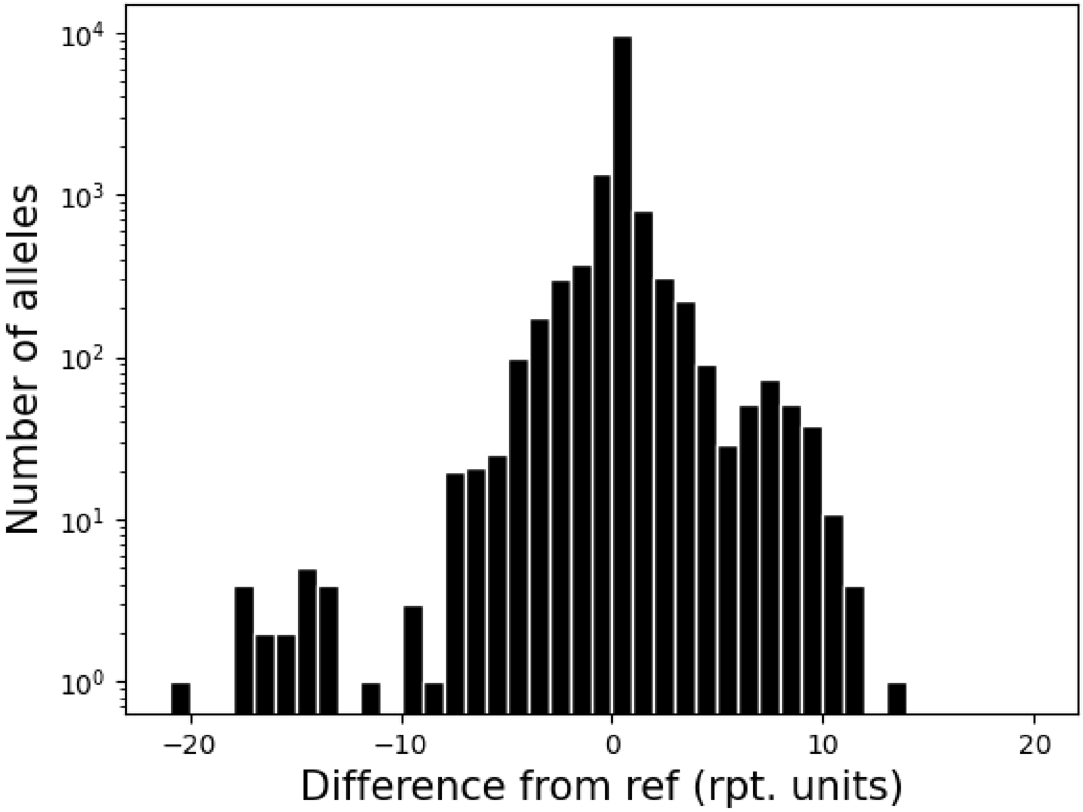
`<outprefix>-sample-callnum.pdf`: a barplot giving the number of calls for each sample. Can be used to determine failed or outlier samples.



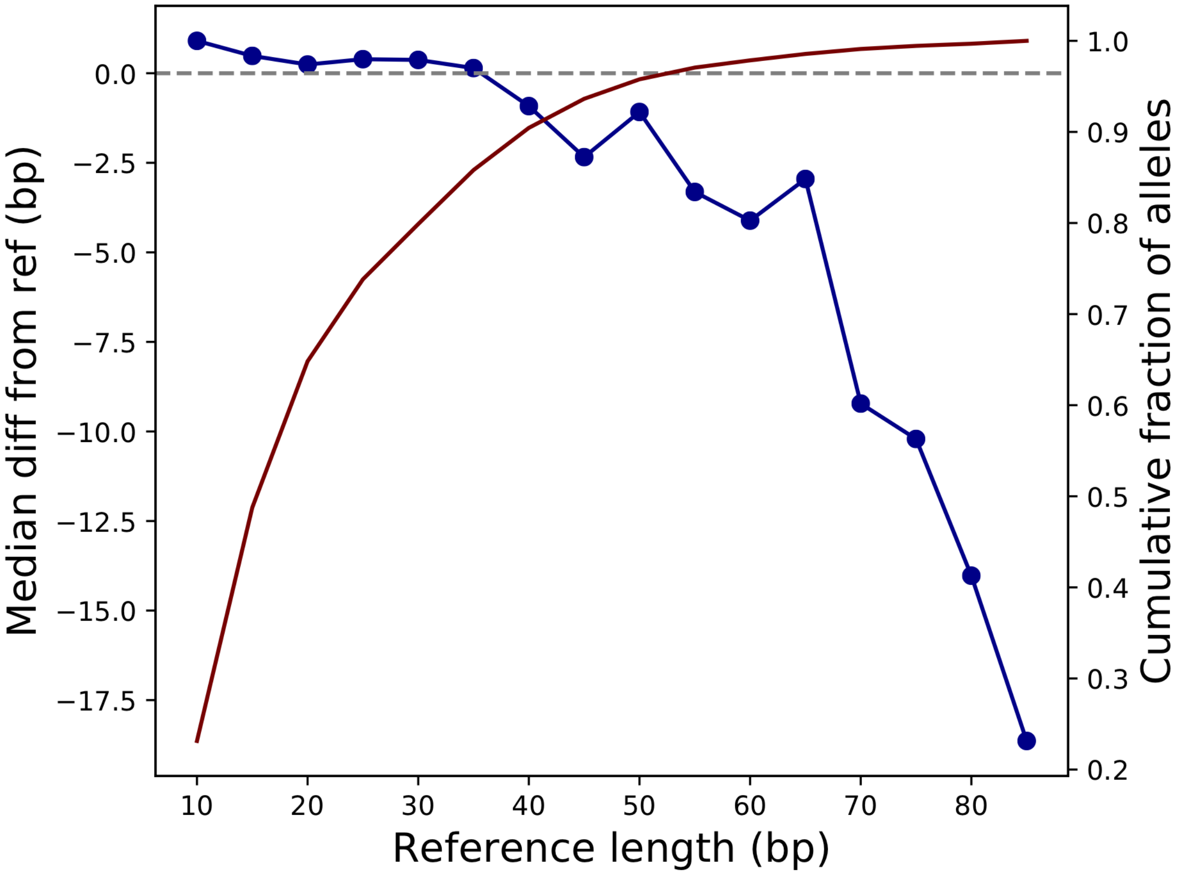
`<outprefix>-chrom-callnum.pdf`: a barplot giving the number of calls for each chromosome. Can be useful to determine if the expected number of calls per chromosome are present.



`<outprefix>-diffref-histogram.pdf`: a histogram of, for each allele called, the difference between its length and the length of the reference at that locus (measured in number of repeat units). Can be used to visualize if there is a strong bias toward calling deletions vs. insertions compared to the reference, which might indicate a problem.



`<outprefix>-diffref-bias.pdf`: plots reference length (bp) vs. the mean (or median) difference in length of each allele called compared to the reference allele. It is expected that the mean difference should be around 0 for most settings. When this value starts to deviate from 0, e.g. for very long repeats, it could indicate a drop in call quality. The red line gives the cumulative fraction of TRs below each reference length.



`<outprefix>-quality.pdf`: plots the cumulative distribution of the quality scores for calls in this VCF. This will be a per-locus plot for >5 samples, or a sample-stratified plot for <= 5 samples. This will not be produced for vcfs which do not have quality metrics. Alternatively, you may specify the type(s) of quality plot(s) you wish to see with the `--quality` option. In that case you will get a file named `<outprefix>-quality-<type>.pdf` for each type of plot you requested. Quality plot examples are shown below. To learn more about how qcSTR infers quality scores for VCFs from different genotypers, see here

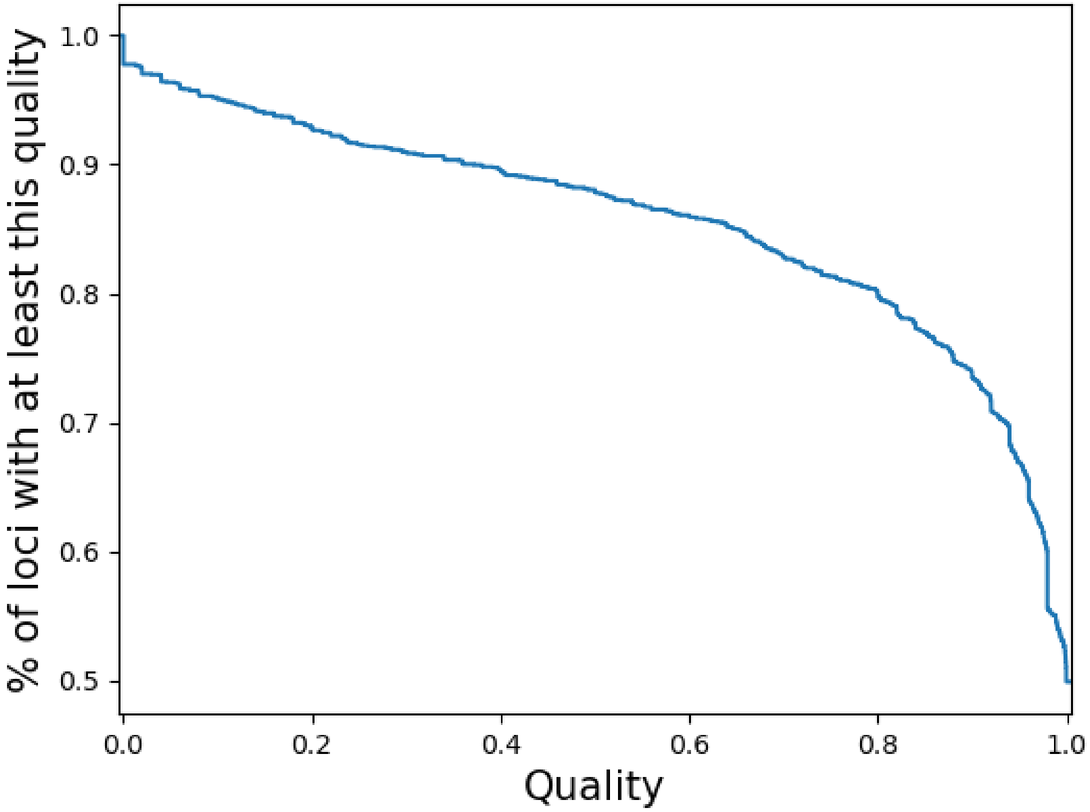
Note: quality score plots are useful when considered in the context of a single genotyper run, and can also be used to compare different invocations of the same genotyper. However, quality score values produced by different genotypers have different meanings due to the different modeling assumptions the different genotypers make. If you wish to compare quality scores across genotypers, you **must** first understand those different assumptions and infer how they will affect the validity of your comparison.

Quality Plot Options

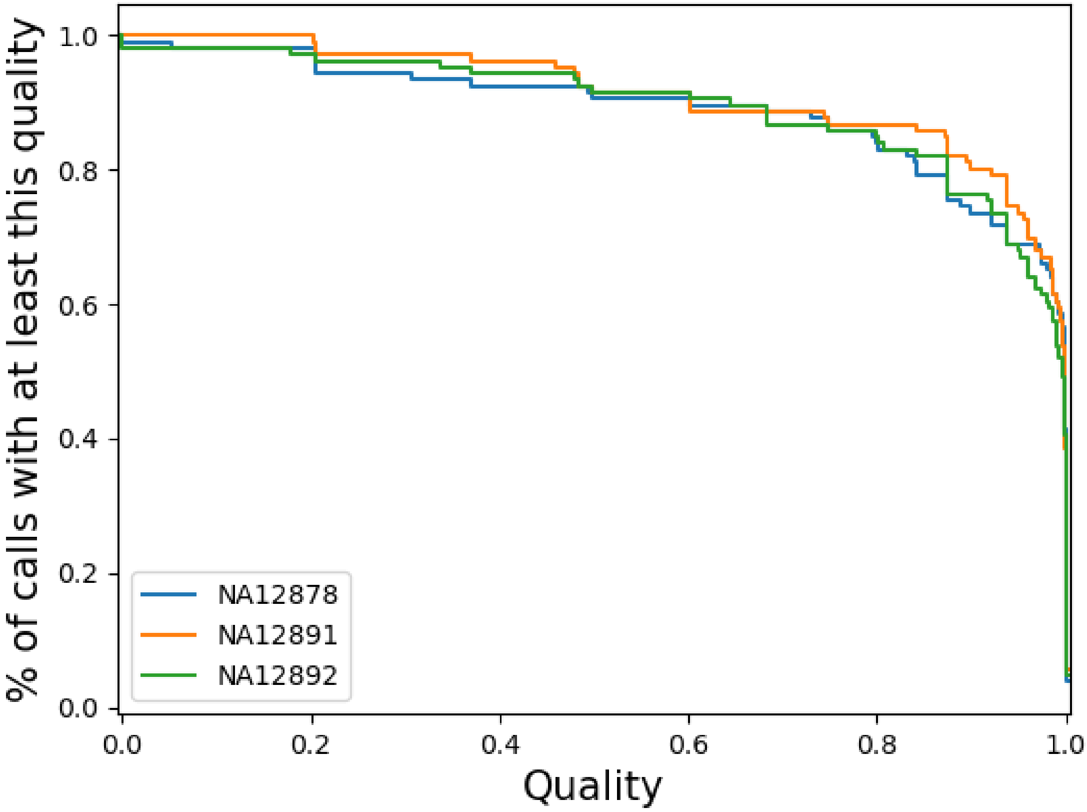
These additional options can be used to customize quality score distribution plots.

`--quality`: This option determines if the plot is stratified and what distribution the y-axis represents. The x-axis is always the quality score and the y-axis is always a decreasing CDF. This can be specified multiple times to produce multiple plots (e.g. `--quality per-locus --quality per-sample`). Each plot will have the specified type appended to the output filename.

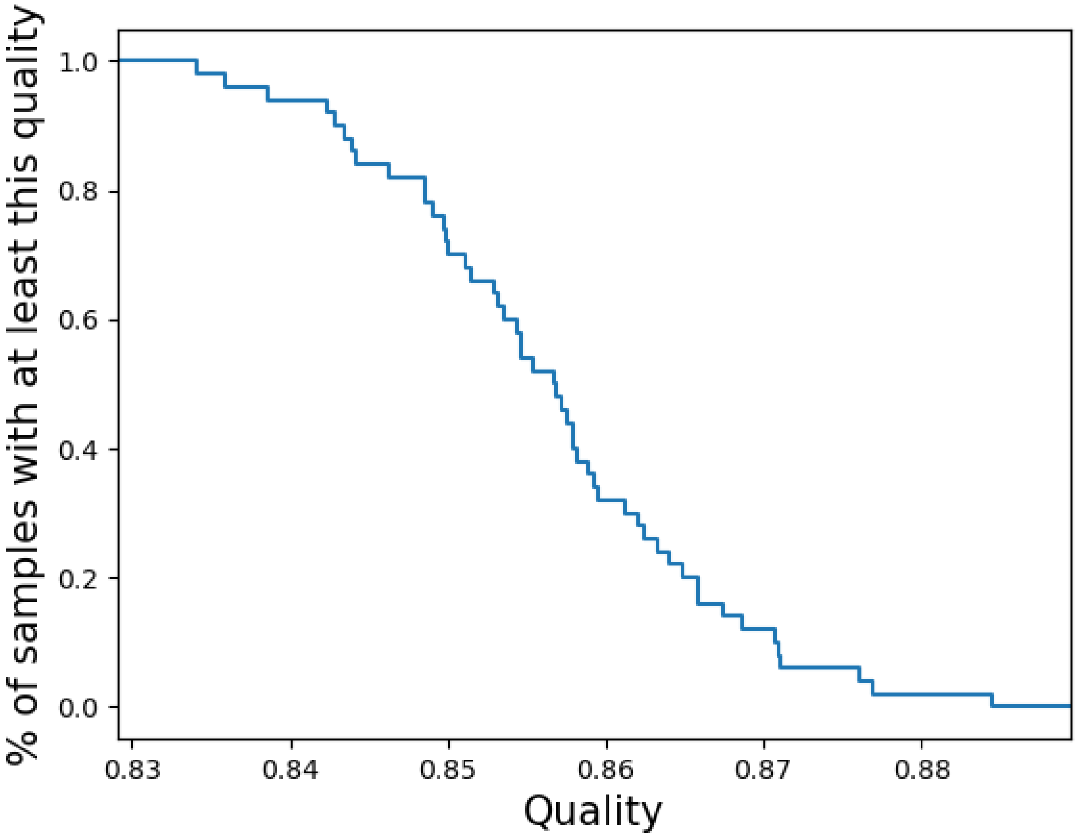
- `per-locus` Compute the call quality at each locus averaged across all samples. Plot the distribution of those loci qualities.



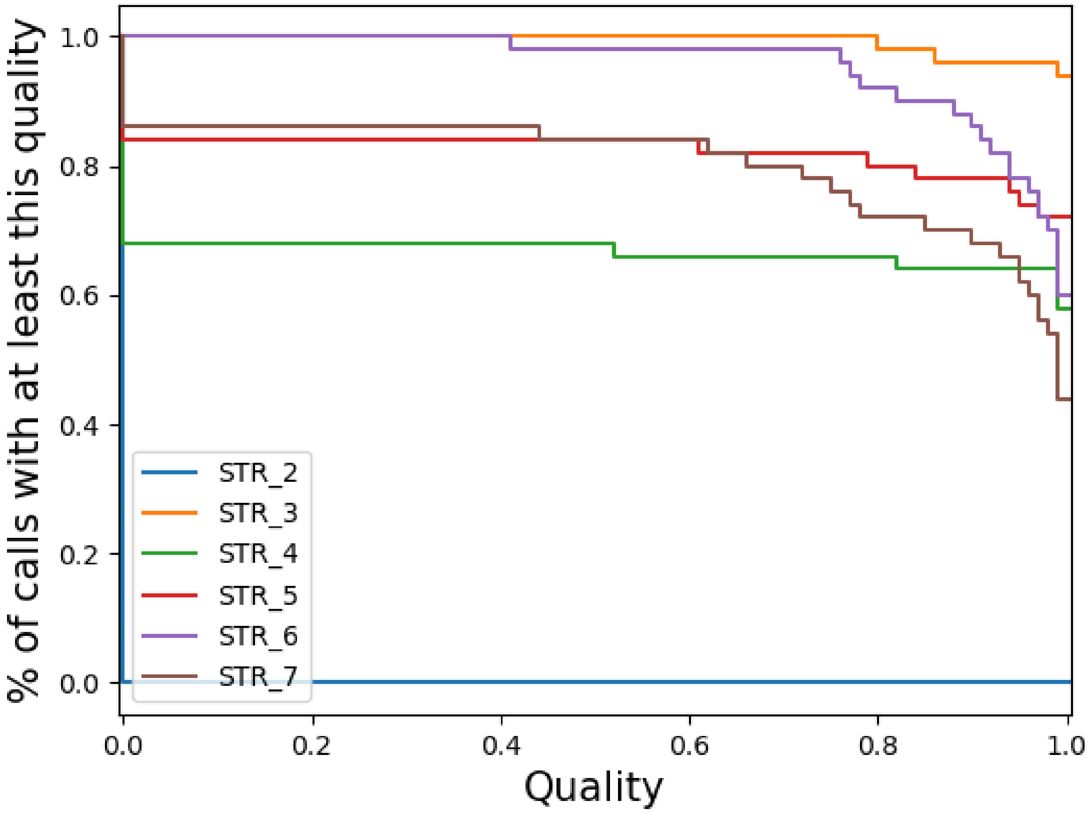
- `sample-stratified` Plot a separate line for each sample of the distribution of loci qualities for that sample. Note: If you specify this for a vcf with many samples, the code may slow to a halt, run out of memory and/or the plot may be cluttered.



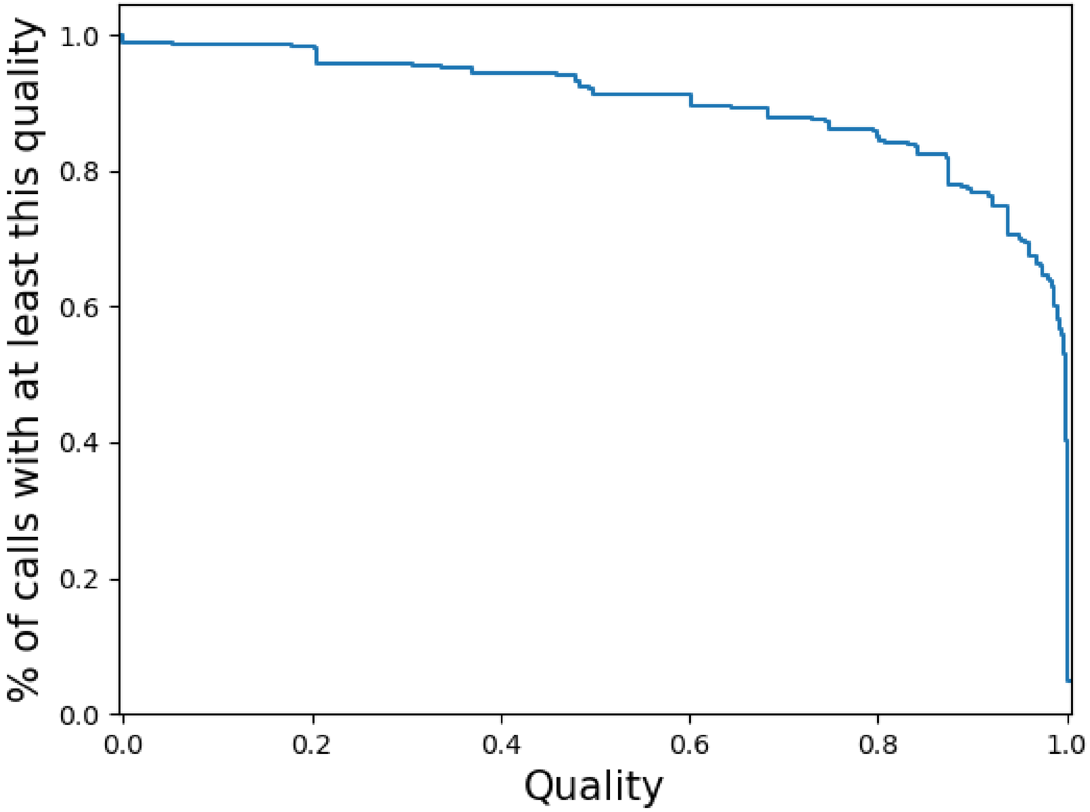
- `per-sample` Compute the call quality for each sample averaged across all loci. Plot the distribution of those sample qualities.



- `locus-stratified` Plot a separate line for each locus of the distribution of sample qualities at that locus. Note: If you specify this for a vcf with many loci, the code may slow to a halt, run out of memory and/or the plot may be cluttered.



- `per-call` Plot the distribution of the quality of all calls. Note: If you specify this for a large vcf the code may run out of memory.



`--quality-ignore-no-call` - by default, (sample, locus) pairs which were not called are treated as calls with zero quality for the the quality plot. With this option enabled, instead they are ignored. This may cause the plotting to crash if it causes some samples/loci to have <= 1 valid call.

Reference Bias Plot Options

These additional options can be used to customize reference bias plots.

- `--refbias-binsize <int>`: Sets the binsize (in bp) used to bin x-axis values, which give the reference TR length. Default=5.
- `--refbias-metric <string>`: Determines which metric to use to summarize the reference bias in each bin. Default=mean. Must be one of: `mean` or `median`.
- `--refbias-mingts <int>`: Exclude points computed using fewer than this many genotypes. This option is meant to avoid plotting outlier points driven by bins with small numbers of TRs with that reference length. Default=100.
- `--refbias-xrange-min <int>`: Exclude points corresponding to TRs with reference length less than this value.
- `--refbias-xrange-max <int>`: Exclude points corresponding to TRs with reference length greater than this value.

Example Commands

Below are qcSTR examples using VCFs from supported TR genotypers. Data files can be found at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

```
# AdVNTR
qcSTR --vcf NA12878_chr21_advntr.sorted.vcf.gz --out test_qc_advntr

# ExpansionHunter
qcSTR --vcf NA12878_chr21_eh.sorted.vcf.gz --out test_qc_eh

# GangSTR
qcSTR --vcf trio_chr21_gangstr.sorted.vcf.gz --out test_qc_gangstr --period 4 --quality␣
↪per-locus

# HipSTR
qcSTR --vcf trio_chr21_hipstr.sorted.vcf.gz --out test_qc_hipstr --vcftype hipstr --
↪samples ex-samples.txt

# PopSTR
qcSTR --vcf trio_chr21_popstr.sorted.vcf.gz --out test_qc_popstr
```

10.2.4 CompareSTR

CompareSTR compares different TR callsets generated on the same samples against the same reference panel. CompareSTR outputs overall, per-locus, and per-sample concordance information.

Example use cases include:

- Comparing calls to a "ground truth" set, e.g. from capillary electrophoresis data, to find call errors
- Comparing calls for the same tool using different parameter settings to identify differences due to bioinformatic processing
- Comparing calls for different tools. This only works if they used the same set of reference TRs. Please note that compareSTR only compares TRs with matching chromosomes and allele-positions (after ignoring flanking base pairs)

CompareSTR optionally will stratify results based on a user-specified FORMAT field (e.g. depth, or quality score) and/or by repeat motif length.

Note: CompareSTR is designed to be used as a QC tool. While it may be able to pick up certain biological differences in some applications (e.g. identifying de novo mutations by comparing parent and child callsets or somatic mutations by comparing callsets from different tissues), use-case specific analyses may be better performed by more specialized tools.

Note: CompareSTR has the ability to stratify comparisons based on quality scores. However, beware that quality scores output by different genotypers may not be directly comparable. You can use qcSTR to visualize the distribution of quality scores in each VCF file seprately.

Usage

CompareSTR takes as input two VCF files with overlapping TRs and samples and outputs metrics and plots based on comparing calls across the two VCFs. The input VCF files must be sorted, indexed, and have the appropriate *##contig* header lines. CompareSTR only considers the subset of samples shared across two VCF files being compared, based on sample ID in the VCF headers.

Note: if comparing two VCFs with **phased** calls, ensure the chromosome ordering matches. A 'motherAllele|fatherAllele' representation will not match with a 'fatherAllele|motherAllele' representation.

To run compareSTR use the following command:

```
compareSTR \
  --vcf1 <file1.vcf.gz> --vcf2 <file2.vcf.gz> \
  --out <string> \
  [additional options]
```

Required Parameters:

- `--vcf1 <VCF>`: First VCF file to compare (must be sorted, bgzipped, and indexed. See *Instructions on Compressing and Indexing VCF files* below)
- `--vcf2 <VCF>`: Second VCF file to compare (must be sorted, bgzipped, and indexed. See *Instructions on Compressing and Indexing VCF files* below)
- `--out <string>`: Prefix to name output files

Filtering Options:

- `--samples <string>`: File containing list of samples to include. If not specified, all samples are used.
- `--region <string>`: Restrict to this region chrom:start-end

Metrics to stratify results:

- `--stratify-fields`: Comma-separated list of FORMAT fields to stratify by. e.g. DP,Q
- `--stratify-binsizes`: Comma-separated list of min:max:binsize to stratify each field on. Must be same length as `--stratify-fields`. e.g. 0:50:5,0:1:0.1 . The range [min, max] is inclusive.
- `--stratify-file`: Specify which file to look at the `--stratify-fields` in. If set to 0, apply to both files. If set to 1, apply only to `--vcf1`. If set to 2, apply only to `--vcf2`.
- `--period`: Report results overall and also stratified by repeat unit length (period).

Plotting options:

- `--bubble-min`: Minimum x/y axis value to display on bubble plots.
- `--bubble-max`: Maximum x/y axis value to display on bubble plots.

Other options:

- `--verbose`: Print helpful debugging info
- `--noplot`: Don't output any plots. Only produce text output.
- `--vcftype1 <string>`: Type of VCF file 1.
- `--vcftype2 <string>`: Type of VCF file 2.
- `--ignore-phasing`: Treat all calls as if they are unphased

Outputs

In output files, compareSTR reports the following metrics:

- Length concordance: % of genotypes concordant between the two VCF files when only considering TR allele lengths
- Sequence concordance: % of genotypes concordant between the two VCF files when considering TR allele sequence. Currently only relevant for HipSTR. Otherwise will be identical to length concordance
- R2: Pearson r2 between the sum of allele lengths at each call compared between the two VCF files, where allele lengths are measured as number of repeat copies different from the reference.

These metrics and numcalls only reflect the (sample, locus) pairs that were called by both callers

compareSTR outputs the following text files and plots:

- `<outprefix>-overall.tab`: Has columns period, concordance-seq, concordance-len, r2, numcalls. Plus additional columns for any FORMAT fields to stratify results on. This file has one line for all results (period="ALL") and a different line for each period analyzed separately if request by `--period`. If stratifying by format fields, it will have additional lines for each range of values for each of those fields.
- `<outprefix>-bubble-period$period.pdf`: "Bubble" plot, which plots the sum of allele lengths for each call in `--vcf1` vs. `--vcf2`. Allele lengths are given in terms of difference in number of repeat units from the reference. The size of each bubble gives the number of calls at each cooordinate. A seperate plot is output for all TRs (period="ALL") and for each period if requested by `--period`.
- `<outprefix>-locuscompare.tab`: Has columns chrom, start, metric-conc-seq, metric-conc-len, numcalls. There is one line for each TR.
- `<outprefix>-locuscompare.pdf`: Plots the length concordance metric for each TR locus considered.
- `<outprefix>-samplecompare.tab`: Has columns sample, metric-conc-seq, metric-conc-len, numcalls. One line per sample
- `<outprefix>-samplecompare.pdf`: Plots the length concordance metric for each sample considered.

See *Example Commands* below for example compareSTR commands for different supported TR genotypers based on example data files in this repository. More detailed use cases are also given in the vignettes https://trtools.readthedocs.io/en/stable/VIGNETTES.html

Instructions on Compressing and Indexing VCF files

CompareSTR requires input files to be compressed and indexed. Use the following commands to create compressed and indexed vcf files:

```
bgzip file.vcf
tabix -p vcf file.vcf.gz
```

Example Commands

Below are compareSTR examples using VCFs from supported TR genotypers. Data files can be found at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

```
# AdVNTR (comparing a file against itself. Not very interesting. Just for demonstration)
# Note, you first need to reheader files to add required contig lines to VCF headers
bcftools reheader -f hg19.fa.fai -o NA12878_advntr_reheader.vcf.gz NA12878_chr21_advntr.
↪sorted.vcf.gz
tabix -p vcf NA12878_advntr_reheader.vcf.gz
FILE1=NA12878_advntr_reheader.vcf.gz
compareSTR --vcf1 ${FILE1} --vcf2 ${FILE1} --out advntr_vs_advntr --noplot

# HipSTR vs. ExpansionHunter
compareSTR \
    --vcf1 NA12878_chr21_hipstr.sorted.vcf.gz \
    --vcf2 NA12878_chr21_eh.sorted.vcf.gz \
    --vcftype1 hipstr --vcftype2 eh --out hipstr_vs_eh

# HipSTR vs. GangSTR
compareSTR \
    --vcf1 NA12878_chr21_hipstr.sorted.vcf.gz \
    --vcf2 NA12878_chr21_gangstr.sorted.vcf.gz \
    --vcftype1 hipstr --vcftype2 gangstr --out hipstr_vs_gangstr

# PopSTR (comparing a file against itself. Not very interesting. Just for demonstration)
FILE1=trio_chr21_popstr.sorted.vcf.gz
compareSTR --vcf1 ${FILE1} --vcf2 ${FILE1} --out popstr_vs_popstr
```

10.2.5 MergeSTR

MergeSTR merges multiple VCF files produced by the same TR genotyper into a single VCF file.

If TR genotyping was performed separately on different samples or batches of samples, mergeSTR can be used to combine the resulting VCFs into one file. This is often necessary for downstream steps such as: computing per-locus statistics, performing per-locus filtering, and association testing.

While other VCF libraries have capabilities to merge VCF files, they do not always handle multi-allelic TRs properly, especially if the allele definitions are different across files. MergeSTR is TR-aware. See below for specific VCF fields supported for each genotyper.

mergeSTR does not support merging VCFs produced by different TR genotypers - that is a more complex usecase, and we are designing a separate tool to do that.

Usage

MergeSTR takes as input two or more VCF files with TR genotypes and outputs a combined VCF file. Note, input VCF files must be bgzipped, sorted, indexed, and have the appropriate `##contig` header lines. See *Instructions on Compressing and Indexing VCF files* below for commands for preparing tabix-indexed VCF files.

To run mergeSTR use the following command:

```
mergeSTR \
  --vcfs <file1.vcf.gz,file2.vcf.gz,...> \
  --out <string> \
  [additional options]
```

Required Parameters:

- `--vcfs <VCFs>`: Comma-separated list of VCF files to merge. All must have been created by the same TR genotyper. Must be bgzipped, sorted, and indexed. (See *Instructions on Compressing and Indexing VCF files* below)
- `--vcfs-list <FILE>`: As an alternative to `--vcfs`, you can provide a file with a list of bgzipped/sorted/indexed VCF files (one filename per line) to merge.
- `--vcftype <string>`: Type of VCF files being merged. Default = `auto`. Must be one of: `gangstr`, `advntr`, `hipstr`, `eh`, `popstr`.
- `--out <string>`: prefix to name output files

Special Merge Options:

- `--update-sample-from-file`: Append file names to sample names. Useful if sample names are repeated across VCF files.

Optional Additional Parameters:

- `--verbose`: Prints out extra information
- `--quiet`: Doesn't print out anything

MergeSTR outputs a merged VCF file `$out.vcf` with the merged genotypes. See *Example Commands* below for example mergeSTR commands for different supported TR genotypers based on example data files in this repository.

Note: when running MergeSTR on HipSTR data, MergeSTR first strips the flanking basepairs from all alleles before merging, allowing records with different flanking basepairs but the same repeat to be comparable. Be aware that in the infrequent occurences where there are indels in the flanking regions called by HipSTR this will cause incorrect alleles to be outputted by the merge.

Supported VCF fields

In addition to proper merging of alleles at multi-allelic sites, MergeSTR supports the following VCF fields for each tool. Fields not listed are currently ignored when merging. INFO fields below are expected to be constant across loci being merged.

AdVNTR

- Supported INFO fields: END, RU, RC, VID
- Supported FORMAT fields: DP, SR, FL, ML

ExpansionHunter

- Supported INFO fields: END, REF, REPID, RL, RU, SVTYPE, VARID

- Supported FORMAT fields: ADFL, ADIR, ADSP, LC, REPCI, REPCN, SO

GangSTR

- Supported INFO fields: END, RU, PERIOD, REF, EXPTHRESH
- Supported FORMAT fields: DP, Q, REPCN, REPCI, RC, ENCLREADS, FLNKREADS, ML, INS, STDERR, QEXP

HipSTR

- Supported INFO fields: START, END, PERIOD
- Supported FORMAT fields: GB, Q, PQ, DP, DSNP, PSNP, PDP, GLDIFF, DSTUTTER, DFLANKINDEL, AB, FS, DAB, ALLREADS, MALLREADS

PopSTR

- Supported INFO fields: Motif
- Supported FORMAT fields: AD, DP, PL

Instructions on Compressing and Indexing VCF files

MergeSTR requires the input file to be compressed and indexed. Use the following commands to create compressed and indexed vcf file:

```
bgzip file.vcf
tabix -p vcf file.vcf.gz
```

Example Commands

Below are mergeSTR examples using VCFs from supported TR genotypers. Data files can be found at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

```
# AdVNTR
# Note, you first need to reheader files to add required contig lines to VCF headers
for sample in NA12878 NA12891 NA12892; do
    bcftools reheader -f hg19.fa.fai -o ${sample}_advntr_reheader.vcf.gz ${sample}_chr21_
↪advntr.sorted.vcf.gz
    tabix -p vcf ${sample}_advntr_reheader.vcf.gz
done
FILE1=NA12878_advntr_reheader.vcf.gz
FILE2=NA12891_advntr_reheader.vcf.gz
FILE3=NA12892_advntr_reheader.vcf.gz
mergeSTR --vcfs ${FILE1},${FILE2},${FILE3} --out test_merge_advntr --vcftype advntr --
↪update-sample-from-file # outputs test_merge_advntr.vcf

# ExpansionHunter
# Note, you first need to reheader files to add required contig lines to VCF headers
for sample in NA12878 NA12891 NA12892; do
    bcftools reheader -f hg19.fa.fai -o ${sample}_eh_reheader.vcf.gz ${sample}_chr21_eh.
↪sorted.vcf.gz
    tabix -p vcf ${sample}_eh_reheader.vcf.gz
done
FILE1=NA12878_eh_reheader.vcf.gz
```

(continues on next page)

(continued from previous page)

```
FILE2=NA12891_eh_reheader.vcf.gz
FILE3=NA12892_eh_reheader.vcf.gz
mergeSTR --vcfs ${FILE1},${FILE2},${FILE3} --out test_merge_eh --vcftype eh # outputs␣
↪test_merge_eh.vcf

# GangSTR
FILE1=NA12878_chr21_gangstr.sorted.vcf.gz
FILE2=NA12891_chr21_gangstr.sorted.vcf.gz
FILE3=NA12892_chr21_gangstr.sorted.vcf.gz
mergeSTR --vcfs ${FILE1},${FILE2},${FILE3} --out test_merge_gangstr --vcftype gangstr #␣
↪outputs test_merge_gangstr.vcf

# HipSTR
FILE1=NA12878_chr21_hipstr.sorted.vcf.gz
FILE2=NA12891_chr21_hipstr.sorted.vcf.gz
FILE3=NA12892_chr21_hipstr.sorted.vcf.gz
mergeSTR --vcfs ${FILE1},${FILE2},${FILE3} --out test_merge_hipstr --vcftype hipstr #␣
↪outputs test_merge_hipstr.vcf

# PopSTR
FILE1=NA12878_chr21_popstr.sorted.vcf.gz
FILE2=NA12891_chr21_popstr.sorted.vcf.gz
FILE3=NA12892_chr21_popstr.sorted.vcf.gz
mergeSTR --vcfs ${FILE1},${FILE2},${FILE3} --out test_merge_popstr --vcftype popstr #␣
↪outputs test_merge_popstr.vcf
```

10.2.6 AssociaTR

associaTR performs association testing of the lengths of TRs against phenotypes.

This tool is designed for running a GWAS of TRs against a single outcome. While slower, it can also be invoked once for each phenotype to perform a PHWAS of a group of TRs against many phenotypes. Many tools perform association testing and are more fully featured than associaTR - the main value of associaTR is its ability to conveniently perform length based tests for multiallelic TR loci instead of per-allele based tests.

associaTR is in **beta**. In particular, while its main outputs (p-values, coefficients and standard errors) are all fully tested, it has not been fully code-reviewed, and other outputs still need testing. It also does not currently support binary phenotypes.

Importantly, associaTR works with Beagle dosages, as imputation dosages are more reliable than imputed best-guess genotypes for association testing.

Usage

Required positional parameters:

- `outfile` - the location to write the association results tsv to.
- `tr_vcf` - the location of vcf containing the TR genotypes to test
- `phenotype_name` - the name of the phenotype being tested against, used to write appropriately named column headers

- `traits` - At least one .npy 2d float array file containing trait values for samples. The first trait is the phenotype to be regressed against, all other traits from that file are used as covariates. If more than one file is provided, then all traits in the additional files are added as additional covariates.

  If `--same-samples` is not specified, the first column of each file must be the numeric sample ID, designating which row corresponds to which sample. (Thus the phenotype will correspond to the second column from the first file.) If multiple files are specified, they will be joined on sample ID.

  If `--same-samples` is specified, there must be the same number of rows in each array as the number of samples in the vcf, those will be the traits for those samples. (Thus the phenotype will correspond to the first column of the first array). If multiple files are specified, then they will be concatenated horizontally. Since IDs do not need to be stored in the npy arrays, `--same-samples` allows for non-numeric sample IDs.

Optional input parameters:

- `--vcftype` One of `eh, hipstr, gangstr, popstr, advntr` Specify which caller produced the TR VCF, useful when the VCF is ambiguous and the caller cannot be automatically inferred.
- `--same-samples` - see the description of traits above
- `--beagle-dosages` - regress against Beagle dosages from the AP{1,2} fields instead of from the GT field. Note: The GP field that Beagle outputs is not supported

Optional filtering parameters:

- `--sample-list` File containing list of samples to use, one sample ID per line. If not specified, all samples are used.
- `--region` Restrict to chr:start-end
- `--non-major-cutoff` - If not `--beagle-dosages`, then this is the non-major-allele-count cutoff. I.e. filter all loci with `non-major-allele-count < cutoff.` If working with dosages, this cutoff is applied to the dosage sums. As with the regression itself, for this purpose alleles are coallesced by length. Note that this is a raw value cutoff, not a percentage cutoff. This defaults to 20 per plink's best practices Set to 0 to disable this filter.

Regression details

AssociaTR performs ordinary least squares regression, i.e. it fits the model

> y = g*beta + Xb + error

where *y* is the outcome, *g* are the TR length genotypes, *beta* is the coefficient associating *y* and *g*, *X* are the other covariates, *b* are the coefficients of those covariates, and *error* is a normally distributed error term. associaTR reports the p-value corresponding to the hypothesis `beta != 0`, *beta* and the standard error of *beta*. It also reports the *R^2* value of this linear model.

If not using dosages, then *g* is simply `sum(len(allele_1) + len(allele_2))` for each sample. If using dosages, then *g* is `_{lengths L} L*[Pr(len(allele_1) == L) + Pr(len(allele_2) == L)]` for each sample.

An intercept term is always automatically included in *X* - do not include one yourself. *y*, *g* and *X* are always standardized (transformed to mean zero and variance 1). So it is not necessary for you to pre-standardize *y* and *X*. Coefficients and standard errors are reported in the original units of the provided traits, not on the standardized units. However, if you pre-standardize your traits, then they will by necessity be reported in those standardized units. The coefficient and standard error are reported in change per number of repeat copies. To get them measured in change per basepair, divide by the length of the repeat unit.

Outputs

- `chrom` - chromosome of the TR
- `pos` - start position in bp of the TR
- `alleles` - the lengths of the TR in the input VCF
- `n_samples_tested` - number of samples tested. This is the number of samples in the VCF, restricted by:
  - If `--same-samples` is not specified, then there can be fewer samples in the trait arrays. If so, samples not in those arrays will be omitted.
  - If `--sample-list` is specified, samples not in that list will be omitted.
  - Samples with missing genotypes will be omitted on a per-variant basis.
- `locus_filtered` - False if the locus was not filtered, or a reason for filtering otherwise. Possible reasons:
  - No called samples
  - Only one called length (if dosages, this implies all other lengths have exactly zero dosage)
  - This locus did not pass the optional cutoff specified by `--non-major-cutoff`
  - The number of samples remaining is less than or equal to the number of covariates in the model (including the genotypes and the intercept term).
- `p_{phenotype_name}` - the p-value of the association or nan if the locus was filtered
- `coeff_{phenotype_name}` - the coefficient of the association, measured in phenotype units per repeat copy or nan if the locus was filtered.
- `se_{phenotype_name}` - the standard error of the coefficient or nan if the locus was filtered
- `regression_R^2` - the R^2 of the fitted model or nan if the locus was filtered
- `motif` - the TR's motif
- `period` - the TR's period
- `ref_len` - the length of the reference allele for the TR
- `allele_frequency` - the frequency of each length allele in the tested samples
- `dosage_esimtated_r2_per_length_allele` - Only applicable to `--beagle-dosages`. This is a dictionary of r^2 values, one for each length, correlating each haplotype's imputed probability of obtaining that length vs whether or not the imputed best-guess genotype is that length
- `r2_length_dosages_vs_best_guess_lengths` - Only applicable to `--beagle-dosages`. This is the r^2 value for the correlation between each haplotype's average imputed length (`_{lengths L} L*[Pr(len(allele) == L)`) with the best-guess length `L` of that haplotype.

Example Commands

Below is an `associaTR` example. For this example no TRs causally impact the simulated phenotype. Data files for this example can be found at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

```
associaTR \
  association_results.tsv \
  ceu_ex.vcf.gz \
  simulated_phenotype \
  simulated_traits_0.npy \
```

(continues on next page)

(continued from previous page)

```
  simulated_traits_1.npy \
  --same-samples
```

10.2.7 PrancSTR

prancSTR quantifies evidence of somatic mosaicism at STRs using VCF files generated by HipSTR.

Tool overview

Population-level heterogeneity arises due to germline mutations that occur before the formation of the zygote and are inherited by all cells in the offspring. However, heterogeneity within an individual may also exist due to somatic mutations that occur post-zygotically in only a sub-population of cells.

prancSTR is a tool for detecting somatic mosaicism at STRs using high throughput sequencing data. It is designed to be run downstream of a germline TR genotyper. It currently only supports analysis of VCF files output by HipSTR. Note that prancSTR does not require a matched control sample as input. prancSTR uses the following fields from HipSTR VCFs for detecting mosaicism:

- `GT` is used to obtain estimated diploid repeat lengths
- `MALLREADS` is used to obtain the observed distribution of copy numbers across all reads aligning to a locus.
- Stutter parameters are obtained from `INFRAME_UP`, `INFRAME_DOWN`, and `INFRAME_PGEOM`.

prancSTR is in *beta*.

Usage

To run prancSTR use the following command:

```
prancSTR \
  --vcf <vcf file> \
  --out <string> \
  [filter options]
```

Required parameters:

- `--vcf <VCF>` Input VCF file, generated by HipSTR.
- `--out <string>` Output file prefix. Use `stdout` to print file to standard output

prancSTR will output a tab-delimited file quantifying evidence of mosaicism at each STR either to stdout or to `$out.tab`. See a description of the output file below.

Other general parameters:

- `--region <string>`: Restrict to the region chr:start-end. VCF file must be bgzipped and indexed to use this option.
- `--samples <string>`: Restrict to the given list of samples. Samples are comma separated.
- `--vcftype <string>`: Specify the tool which generated the vcf call file for STRs. Currently this will fail if using anything other than `hipstr` VCFs.
- `--only-passing`: Filters out the VCF records with non-passing FILTER column
- `--output-all`: Force tool to output results for all loci. Overrides :code:`--only-passing`.

- `--readfield <string>`: Specify which VCF format field output by HipSTR to utilize for extracting read information. We recommend setting this to "MALLREADS". "ALLREADS" is also accepted but we have found that it produces unreliable results.
- `--debug`: Print helpful debug messages.
- `--quiet`: Restrict printing of any messages.
- `--version`: Print the version of the tool

Notes:

- For the `--only-passing` option, the FILTER column is generated by dumpSTR or another upstream filtering tool based on user-defined filtering parameters. It is not determined by prancSTR.
- By default, STRs with minimal evidence of mosaicism or suspicious read count fields are skipped to save time. These include loci where reads show evidence of only a single allele or for which the germline alleles called by HipSTR are not actually supported by any reads. To force results to be output for all loci, use the `--output-all` parameter. This overrides `--only-passing`.

See *Example Commands* for examples running prancSTR under different settings.

Output files

The prancSTR output file contains mosaicism predictions generated for each locus. Note, this file contains statistics for all tested loci and it is up to the user's discretion to filter out for high confidence mosaic allele calls. The output generated is a tab-delimited file with one row summarizing evidence of mosaicism for each call analyzed, with the following columns:

- `sample`: The ID of the sample being considered.
- `chrom`: Chromsome of the STR being considered
- `locus`: Reference ID for the short tandem repeat.
- `motif`: The nucleotide sequence of the repeat unit.
- `A`: The first germline allele for the given STR in repeat units relative to the reference (copied from HipSTR).
- `B`: The second germline allele for the given STR in repeat units relative to the reference (copied from HipSTR)
- `C`: Candidate mosaic allele inferred by prancSTR in repeat units relative to the reference (inferred by prancSTR).
- `f`: Estimated mosaic allele fraction (inferred by prancSTR).
- `pval`: Gives the p-value testing the null hypothesis that f=0.
- `reads`: Gives representation for how many reads support each allele (copied from HipSTR VCF field corresponding to the specified `--readfield`).
- `mosaic_support`: The number of reads that support the mosaic allele.
- `stutter parameter u`: The probability that stutter error causes an increase in obs. STR size (copied from HipSTR INFRAME_UP field).
- `stutter parameter d`: The probability that stutter error causes a decrease in obs. STR size (copied from HipSTR INFRAME_DOWN field).
- `stutter parameter rho`: Parameter for geometric step size distribution of stutter errors (copied from HipSTR INFRAME_PGEOM field).
- `quality factor`: Quality score of the germline genotype (copied from HipSTR Q field).
- `read depth`: Reports the total depth/number of informative reads for all samples at the locus (copied from HipSTR DP field).

Below shows several example output lines from running prancSTR:

sample	chrom	pos	locus	motif	A	B	C	f	pval	reads	mosaic_support	stutter parameter u	stutter paramter d	stutter paramter rho	quality factor	read depth
NA07022	chr1	987287	Human_STR_285	T	3	5	2	0.24407	1.9530865e-04	3|4;4|4;6|1	5	0.01	0.07	0.31	0.98	21
NA12716	chr1	987287	Human_STR_285	T	5	5	4	0.26595	1.8424416e-05	4|6;5|16;9|1	5	0.01	0.07	0.31	1.00	34
NA06989	chr1	1002414	Human_STR_295	T	-1	4	3	0.15096	1.0045544e-05	-1|16;3|4;4|9	4	0.02	0.02	0.69	1.00	50
NA10847	chr1	1002414	Human_STR_295	T	4	5	3	0.28068	1.290112e-09	3|7;4|6;5|11	7	0.02	0.02	0.69	1.00	55
NA12347	chr1	1002414	Human_STR_295	T	5	5	4	0.26235	1.029537e-05	4|5;5|14;6|1	5	0.02	0.02	0.69	0.99	51

As a starting point, we suggest filtering output on the following parameters to obtain candidate mosaic sites:

- `pval`:of less than or equal to 0.05/(number of STRs tested). The number of STRs tested is equal to the number of data lines in the prancSTR output file.
- `read depth`: of greater than or equal to 10
- `quality factor` of greater than or equal to 0.8
- `mosaic_support` of greater than or equal to 3
- `f`: of less than equal to 0.3. Higher f values are often indicative of a heterozygous genotype miscalled as homozygous.

Example Commands

Below are `prancSTR` examples using HipSTR VCFs. Data files can be found at https://github.com/gymrek-lab/TRTools/tree/master/example-files:

```
# Example command running prancSTR for only one chromosome with hipstr output file
# --only-passing skips VCF records with non-passing filters
prancSTR \
   --vcf example-files/CEU_subset.vcf.gz \
   --out CEU_chr1  \
   --vcftype hipstr \
   --only-passing \
   --region chr1

# Example command running prancSTR for only one sample
# --only-passing skips VCF records with non-passing filters
prancSTR \
   --vcf example-files/CEU_subset.vcf.gz \
   --only-passing \
   --out NA12878_chr1 \
   --samples NA12878
```

Citations

The latest manuscript for citation of the tool can be found at https://doi.org/10.1101/2023.11.22.568371

10.2.8 SimTR

SimTR simulates next generation sequencing reads at a single TR region while modeling stutter errors common in such sequencing.

Prerequisites

simTR is a wrapper on the Illumina ART tool. To run simTR, either the command `art_illumina` must be in a directory on your `PATH`, or alternatively you can specify the path to the ART executable using the `--art` option. If you installed simTR via conda, ART should be installed already.

Usage

To run simTR use the following command:

```
simTR \
  --ref <fasta file> \
  --coords <chr:start-end> \
  --repeat-unit <str> \
  --outprefix <str>
  [additional options]
```

Required parameters:

- `--ref`: Path to the reference fasta file
- `--coords`: The coordinates of the TR from which to simulate reads. Format: chrom:start-end.
- `--repeat_unit`: The sequence of the repeated unit.
- `--outprefix`: Prefix to name output files.

By default, `simTR` will simulate paired end reads and output reads to `$outprefix_1.fq` and `$outprefix_2.fq`. Options described below allow for changing the error model or sequencing parameters.

Stutter model

Insertions or deletions of repeat units (commonly referred to as stutter errors) are simulated according to the model specified in the HipSTR manuscript. The model can be specified using three optional parameters:

- `--u <float>`: Probability a read contains stutter error that increases the total number of repeat units (Default: 0.05)
- `--d <float>`: Probability a read contains stutter error that decreases the total number of repeat units (Default: 0.05)
- `--rho <float>`: The step size parameter. Stutter error sizes are drawn from a geometric distribution with parameter rho (Default: 0.9)
- `--p-thresh <float>`: Ignore stutter alleles expected to have less than this frequency (Default: 0.001)
- `--seed <int>`: Set the random seed for reproducibility.

Sequencing options

You can specify the following sequencing options:

- `--single`: Simulate single end reads. By default, paired end reads are output.
- `--coverage <int>`: Target coverage to simulate (ART parameter `-f`). (Default: 1000)
- `--read-length <int>`: Read length (ART parameter `-l`). (Default: 100)
- `--insert <float>`: Mean fragment length for paired end reads (ART parameter `-m`). (Default: 350)
- `--sd <float>`: Standard deviation of the fragment length for paired end reads (ART parameter `-s`). (Default: 50)
- `--window <int>`: Window size (bp) around the target TR to simulate. (Default: 1000)

Additional options

Users can optionally specify the following:

- `--tmpdir <str>`: store temporary files within this directory. Otherwise, a folder in `$TMPDIR` is created. Temporary files include "dummy" fasta files with different TR alleles and intermediate fastqs generated by ART.
- `--art <str>`: path to the `art_illumina` executable.

Known issues

- Currently requires repeat boundaries to be exact perfect copies. Could instead infer the rotation of the repeat unit to be more robust to this.

Example Commands

Below is a `simTR` example using an example fasta file, which can be found in the directory `example-files`. Example command:

```
# Example command running simTR for a dummy dataset with dummy allele bed file and other input parameters
mkdir test-simtr
simTR \
   --coords chr11_CBL:5001-5033 \
   --ref example-files/CBL.fa \
   --tmpdir test-simtr \
   --repeat-unit CGG \
   --art art_illumina \
   --outprefix test-simtr \
   --coverage 1000 \
   --read-length 150 \
   --seed 12345 \
   --u 0.02 --d 0.02 --rho 0.9
```

This command should output the files `test-simtr_1.fq` and `test-simtr_2.fq`.

Citations

A preprint describing simTR and prancSTR is currently being prepared.

DumpSTR

- DumpSTR filters VCF files with TR genotypes, performing call-level and locus-level filtering, and outputs a filtered VCF file.

StatSTR

- StatSTR takes in a TR genotyping VCF file and outputs per-locus statistics.

qcSTR

- qcSTR generates plots that are useful for diagnosing issues in TR calling.

CompareSTR

- CompareSTR compares different TR callsets generated on the same samples against the same reference panel. CompareSTR outputs overall, per-locus, and per-sample concordance information.

MergeSTR

- MergeSTR merges multiple VCF files produced by the same TR genotyper into a single VCF file.

AssociaTR

- associaTR performs association testing of the lengths of TRs against phenotypes.

PrancSTR

- prancSTR quantifies evidence of somatic mosaicism at STRs using VCF files generated by HipSTR.

SimTR

- SimTR simulates next generation sequencing reads at a single TR region while modeling stutter errors common in such sequencing.

10.3 Supported TR Genotypers

TRTools currently supports 5 tandem repeat genotypers. It also supports the Beagle imputation software (see *below*). We summarize them in the first table and provide some basic parameters of their functionality in the second. For more information on a genotyper, please see its website linked below.

Genotyper (version tested)	Use case notes
AdVNTR (v1.3.3)	Infers allele lengths. May alternatively identify putative frameshift mutations within VNTRs (6+bp repeat units). Designed for targeted genotyping of VNTRs. May be run on large panels of TRs but is compute-intenstive.
ExpansionHunter (v3.2.2)	Handles repeats with structures such as interruptions or nearby repeats. Designed for targeted genotyping of expansions at known pathogenic TRs but may be run genome-wide on short and expanded TRs using a custom TR panel.
GangSTR (2.4.4)	Designed for genome-wide genotyping of short or expanded TRs.
HipSTR (v0.6.2)	Designed for genome-wide genotyping of STR (1-6bp repeat units) alleles shorter than the read length. Can phase repeats with SNPs.
PopSTR (v2.0)	Designed for genome-wide genotyping of short or expanded TRs.

Genotyper (version tested)	Repeat unit lengths	Alleles longer than reads?	Allele type inferred	# TRs in reference	Sequencing technology	# Samples at a time
AdVNTR (v1.3.3)	6-100bp	No	Length, frameshifts	158,522 (genic hg19)	Illumina, PacBio	Single
ExpansionHunter (v3.2.2)	1-6bp. Can handle complex repeat structures specified by regular expressions	Yes	Length	25 (hg19)	PCR-free Illumina	Single
GangSTR (2.4.4)	1-20bp	Yes	Length	829,233 (hg19)	Paired-end Illumina	Many
HipSTR (v0.6.2)	1-9bp	No	Length, sequence	1,620,030 (hg19)	Illumina	Many
PopSTR (v2.0)	1-6bp	Yes	Length	540,1401 (hg38)	Illumina	Many

Since each of these tools take as input a list of TRs to genotype, they could also be used on custom panels of TR loci. Tool information and reference panel numbers shown above are based on downloads from the github repository of each tool as of July 2, 2020.

TRTools can be extended to support other genotypers that generate VCF files. We welcome community contributions to help support them. If that interests you, please see *Contributing* for more information.

10.3.1 Beagle

The Beagle software can take genotypes called by a TR genotyper in a set of reference samples and impute them into other samples that do not have directly genotyped TRs. TRTools supports TR genotypes produced by any of the above genotypers and then imputed into other samples with Beagle except for PopSTR genotypes. For each tool in this tool suite, unless it's docs specifically say otherwise, that tool can be used on Beagle VCFs as if those VCFs were produced directly by the underlying TR genotyper, with no additional flags or arguments needed, as long as the steps below were followed to make sure the Beagle VCF is properly formatted.

Caveats:

- Beagle provides phased best-guess genotypes for each imputed sample at each TR locus. When run with the `ap` or `gp` flags Beagle will also output probabilities for each possible haplotype/genotype, respectively. These probabilities are also called dosages. While dosages are often more informative for downstream analyses than the best-guess genotypes located in the `GT` format field (for instance, for association testing), TRTools currently does *not* support dosage based analyses and instead will only look at the `GT` field. Feel free to submit PRs with features that handle dosages (see the *Contributing* docs).
- At each locus Beagle returns the most probable phased genotype. This will often but not always correspond to the most probable unphased genotype. For instance, it is possible that `P(A|A) > P(A|B)` and `P(A|A) > P(B|A)`, but `P(A/A) = P(A|A) < P(A|B) + P(B|A) = P(A/B)`. Similarly, it is possible that `P(A|B) > P(C|D)` and `P(A|B) > P(D|C)`, but `P(A/B) = P(A|B) + P(B|A) < P(C|D) + P(D|C) = P(C/D)`. TRTools currently does not take this into account and just uses the phased genotypes returned by Beagle. If you deem this to be an issue, feel free to submit PRs to help TRTools take this into account (see the *Contributing* docs).
- For callers which return sequences, not just lengths (e.g. HipSTR), if there are loci with multiple plausible sequences of the same length, then its possible that the most probable genotype returned by Beagle does not have

the most probable length. For example, the following could be true of a single haplotype: `Len(S_1) = L_1, Len(S_2) = L_1, Len(S_3) = L_2` and `P(S_1) < P(S_3), P(S_2) < P(S_3)` but `P(S_3) < P(S_1) + P(S_2)`.

An overview of steps to perform before Beagle imputation:

- The samples being imputed into must have directly genotyped loci that are also genotyped in the reference samples. This allows those samples to be 'matched' with samples in the reference.
- The genotypes of both the reference samples and samples of interest must be phased. That can be done by statistically phasing the genotypes prior to running Beagle imputation.
- The referece samples must also not contain any missing genotypes. Possible methods for dealing with that include removing loci with missing genotypes or using imputation to impute the missing genotypes prior to imputing the TRs.

The VCFs that Beagle outputs need to be preprocessed before use by TRTools. We have provided a tool `trtools_prep_beagle_vcf.sh` to run on those VCFs. After running this script, the files should be usable by any of the tools in TRTools.

In case of error, it may be useful to know what steps the script attempts to perform:

- It copies over source and command meta header lines from the reference panel to the imputed VCF so that it is clear which genotyper's syntax is being used to represent the STRs in the VCF.
- It copies over contig and ALT lines which is required for downstream tools including mergeSTR and is good practice to include in the VCF header.
- It annotates each STR with the necessary INFO fields from the reference panel that Beagle dropped from the imputed VCF.
- The imputed VCF contains both TR loci and the shared loci (commonly SNPs) that were used for the imputation. This script removes the non-STR loci (identified as those loci not having STR-specific INFO fields).

10.4 Python Library

The TRHarmonizer python library underlies all of the TRTools command line tools and can be used independently of them. See *here* for a description of what that library does. See *here* for examples on how to use it in your own code. See *here* for specifics on how TRHarmonizer parses the outputs of different genotypers. And see *here* for the API of the python code of both the TRHarmonizer library and the command-line tools.

10.4.1 The TRHarmonizer Library

The TRHarmonizer Python library provides a uniform interface for accessing VCFs created by different tandem repeat (TR) genotypers. This library is the shared basis for all the command-line tools in the TRTools package. It is designed to cleanly handle differences in how different genotypers represent alleles, quality-scores and other metadata describing TR genotypes. This allows coding against a uniform interface while analyzing genetic variation at TRs regardless of which genotyper was used. The TRHarmonizer library also allows third parties to leverage the harmonization functionality outside of the command-line tools provided in TRTools.

A major challenge in analyzing TR genotypes is that alleles are represented differently in VCF outputs of different genotypers. The example below for chr21:47251618 (hg19) genotyped in Platinum Genomes sample NA12878 shows the different ways reference and alternate alleles are specified in VCFs by the genotypers which TRTools currently supports.

- adVNTR*, GangSTR, HipSTR:
  - REF: AGTTAGTTAGTTAGTT

  - ALT: AGTTAGTTAGTTAGTTAGTT
- ExpansionHunter:
  - REF: A
  - ALT: <STR5>
  - INFO: REF=4;RU=AGTT
- PopSTR:
  - REF: AGTTAGTTAGTTAGTT
  - ALT: <5>
  - INFO: Motif=AGTT

* Note that while this TR was not called by AdVNTR because its motif is too short, AdVNTR output represents alleles in the same format as HipSTR and GangSTR.

Furthermore, consider the example at chr21:16402147

- adVNTR, GangSTR, HipSTR:
  - REF: AAATAAATAAATAAATAAAT
  - ALT: AAATAAATAAATAAAT
- ExpansionHunter:
  - REF: A
  - ALT: <STR4>
  - INFO: REF=5;RU=AAAT
- PopSTR:
  - REF: AAATAAATAAATAAATAAATAATAAA
  - ALT: <5.5>
  - INFO: Motif=AAAT

Here we see that popSTR's representation of alleles changes to specify impurities and partial repeats.

The key function of the TRHarmonizer module, HarmonizeRecord, takes as input a cyvcf2[1] record (a cyvcf2.Variant object) and a VCF type (one of: "advntr", "eh", "gangstr", "hipstr" or "popstr", corresponding to the supported genotypers) and outputs a TRRecord object storing alleles and other metadata in a standardized format. This allows downstream analyses to proceed agnostic of the genotyper which created the record. The TRRecord stores allele length genotypes as the number of copies of the motif corresponding to that length. This number is a float to allow for impurities and partial repeats. For genotypers which infer sequence alleles, the record additionally stores the sequence of the allele in all uppercase. In addition to alleles, a TRRecord also provides a uniform method for accessing the TR motif, per-sample quality scores and other metadata supplied by the underlying genotyper.

TRHarmonizer is designed to be lightweight, and as such there are similar yet more complex use-cases that TRHarmonizer intentionally does not support. It does not have any insight into sequencing technologies which produce data that is later processed by TR genotypers into VCFs. As such it relies on the alleles, calls and associated quality scores output by the genotypers, each of which use their own models to compute quality scores. TRTools makes no attempt to modify those scores based on sequencing errors or other sources of error.

TRHarmonizer also does not handle differences in variant coordinates, whether due to differences in choice of variant reference set or differences between calling algorithms. Note that this is only relevant to compareSTR, as that is the

[1] Brent S Pedersen, Aaron R Quinlan, cyvcf2: fast, flexible variant analysis with Python, Bioinformatics, Volume 33, Issue 12, 15 June 2017, Pages 1867-1869, https://doi.org/10.1093/bioinformatics/btx057

only one of our tools designed to process TRs from multiple VCFs produced by different genotypers simultaneously. The types of differences related to variant coordinates that TRHarmonizer does not handle includes:

- Repeat regions which some callers choose to represent as a single variant and other callers represent as multiple variants
- Overlapping variants of different lengths due to decisions about whether to phase the repeat variant with other nearby variants
- Overlapping variants of different lengths due to different choices as to which parts of a locus constitute impure repeats and which constitute flanking regions

Rather, TRHarmonizer restricts itself to comparing variants called by different callers whose reference alleles start and end at the same base pairs. Handling different variant representations is a complex problem that has been the subject of significant work[2][3] and is best handled by haplotype comparison tools which have been tailored to the specific use-case at hand. If there are compelling use-cases specific to TRs that are not handled by prior work then TRHarmonizer could be extended to tackle them, but that is currently out of scope.

Finally, TRHarmonizer can be readily extended to support any TR genotyping tool built on top of any sequencing or genotyping technology as long as the tool produces a valid VCF file representing each TR as a distinct record in the VCF. Supporting additional tools simply requires adding a short function to the TRHarmonizer module converting records to the standardized format described above.

10.4.2 Code Examples

In addition to command-line utilities, TRTools has a Python library which can be imported in custom scripts.

TR Utilities

The module `trtools.utils.utils` contains various helpful functions, e.g. for inferring a repeat sequence motif from a string, converting repeat motifs to canonical representations, or computing functions like heterozygosity from allele frequency distributions:

```
import trtools.utils.utils as trutils
trutils.InferRepeatSequence('ATATATATATATA', 2) # returns 'AT'
trutils.GetCanonicalMotif('CAG') # returns 'ACG'
afreqs = {10:0.25, 11:0.5, 12: 0.25} # frequency of each length
trutils.GetHeterozygosity(afreqs) # returns 0.625
```

See *here* for a complete list of utility functions.

TR Harmonization

Note: the interfaces provided by this library are still under active development. While their rough shape will likely stay the same, particular details are likely to change in future releases.

The module *trtools.utils.tr_harmonizer module* is responsible for providing a genotyper agnostic view of a VCF containing TR records. The class that provides this functionality is `trtools.utils.tr_harmonizer.TRRecord`. There are two coding paradigms for accessing this API. If you just want to iterate through the TRRecords in a vcf, use the TRRecordHarmonizer:

[2] Cleary, John G., et al. "Comparing variant call files for performance benchmarking of next-generation sequencing variant calling pipelines." BioRxiv (2015): 023754.

[3] https://github.com/Illumina/hap.py

```
import cyvcf2
import trtools.utils.tr_harmonizer as trh

invcf = cyvcf2.VCF("path/to/my.vcf")
harmonizer = trh.TRRecordHarmonizer(invcf)
for trrecord in harmonizer:
      # do something with the trrecord
      # here, we print out an array of length genotypes
      # containing entries for each haplotype of each sample
      print(trrecord.GetLengthGenotypes())
```

If you want to work with the cyvcf2 VCF object yourself and then later convert it to a TRRecord, use the module method HarmonizeRecord:

```
import cyvcf2
import trtools.utils.tr_harmonizer as trh

invcf = cyvcf2.VCF("path/to/my.vcf")
#make sure to grab the vcf's filetype
vcftype = trh.InferVCFType(invcf)
for record in invcf:
  # do some filtering on the record
  passesFilters = filter(record)

  # later on, convert the record to a trrecord
  if not passesFilters:
     continue

  trrecord = trh.HarmonizeRecord(vcftype, record)
  # do something with the tr record
  # here, we print out an array of string genotypes
  # containing entries for each haplotype of each sample
  print(trrecord.GetStringGenotypes())
```

See *trtools.utils.tr_harmonizer* for a complete list of all functions used for inspecting VCFs and TRRecords.

10.4.3 TRHarmonizer Library Details

Here are details about how the harmonizer library (and the TRTools which are built on top of it) handle VCFs produced by the different supported genotypers.

Alleles

Different callers will call alleles differently (sequences or just lengths, if sequences with or without impurities such as partial repeats or internal SNPs, if sequences with or without flanking SNPs)

- AdVNTR - Calls sequence genotypes with impurities.
- ExpansionHunter - Calls lengths genotypes.
- GangSTR - Calls sequence genotypes without impurities.

- HipSTR - Calls sequence genotypes with impurities. Can call flanking base pairs and SNPs, if so TRRecord will have `trtools.utils.tr_harmonizer.TRRecord.GetFullStringGenotypes()`. HipSTR does not report the underlying motif, so it is inferred from the sequence, which is usually but not always correct.
- PopSTR - Calls the reference as a sequence genotype with impurities. Calls alt alleles as length genotypes.

Quality Scores

Quality scores are numbers associated with each call in a VCF where a higher number means a better call. The reliability of these numbers is dependent on the genotyper that emitted them. Here is how TRHarmonizer infers quality scores for each supported genotyper:

- AdVNTR quality scores are taken directly from the `ML` (maximum likelihood) format field.
- ExpansionHunter does not output a quality score. It does output a confidence interval, but this field is not currently used by TRTools.
- GangSTR quality scores are taken directly from the `Q` format field.
- HipSTR quality scores are taken directly from the `Q` format field. TRHarmonizer ignores the `PQ` field, which represents the quality of a call and its phasing.
- PopSTR does not output a genotype quality score. It does output a `PL` field with Phred-scaled genotype likelihoods, but this field is not currently used by TRTools.

More details about these fields can be found in the documentation for each genotyper.

10.4.4 Python API

trtools.associaTR module

class `trtools.associaTR.`**`OLS`**(*endog*, *exog=None*, *missing='none'*, *hasconst=None*, ***kwargs*)

Bases: `statsmodels.regression.linear_model.WLS`

A simple ordinary least squares model.

Parameters

- **endog** (*array-like*) – 1-d endogenous response variable. The dependent variable.
- **exog** (*array-like*) – A nobs x k array where *nobs* is the number of observations and *k* is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant()`.
- **missing** (*str*) – Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’
- **hasconst** (*None or bool*) – Indicates whether the RHS includes a user-supplied constant. If True, a constant is not checked for and k_constant is set to 1 and all result statistics are calculated as if a constant is present. If False, a constant is not checked for and k_constant is set to 0.

weights

Has an attribute weights = array(1.0) due to inheritance from WLS.

Type scalar

See also:

`GLS`

Examples

```
>>> import numpy as np
>>>
>>> import statsmodels.api as sm
>>>
>>> Y = [1,3,4,5,2,3,4]
>>> X = range(1,8)
>>> X = sm.add_constant(X)
>>>
>>> model = sm.OLS(Y,X)
>>> results = model.fit()
>>> results.params
array([ 2.14285714,  0.25      ])
>>> results.tvalues
array([ 1.87867287,  0.98019606])
>>> print(results.t_test([1, 0]))
<T test: effect=array([ 2.14285714]), sd=array([[ 1.14062282]]), t=array([[ 1.
↪87867287]]), p=array([[ 0.05953974]]), df_denom=5>
>>> print(results.f_test(np.identity(2)))
<F test: F=array([[ 19.46078431]]), p=[[ 0.00437251]], df_denom=5, df_num=2>
```

Notes

No constant is added by the model unless you are using formulas.

_fit_ridge(*alpha*)

Fit a linear model using ridge regression.

Parameters alpha (*scalar or array-like*) – The penalty weight. If a scalar, the same penalty weight applies to all variables in the model. If a vector, it must have the same length as *params*, and contains a penalty weight for each coefficient.

Notes

Equivalent to fit_regularized with L1_wt = 0 (but implemented more efficiently).

fit_regularized(*method='elastic_net'*, *alpha=0.0*, *L1_wt=1.0*, *start_params=None*, *profile_scale=False*, *refit=False*, ***kwargs*)

Return a regularized fit to a linear regression model.

Parameters

- **method** (*string*) – 'elastic_net' and 'sqrt_lasso' are currently implemented.
- **alpha** (*scalar or array-like*) – The penalty weight. If a scalar, the same penalty weight applies to all variables in the model. If a vector, it must have the same length as *params*, and contains a penalty weight for each coefficient.
- **L1_wt** (*scalar*) – The fraction of the penalty given to the L1 penalty term. Must be between 0 and 1 (inclusive). If 0, the fit is a ridge fit, if 1 it is a lasso fit.
- **start_params** (*array-like*) – Starting values for `params`.

- **profile_scale** (*bool*) – If True the penalized fit is computed using the profile (concentrated) log-likelihood for the Gaussian model. Otherwise the fit uses the residual sum of squares.
- **refit** (*bool*) – If True, the model is refit using only the variables that have non-zero coefficients in the regularized fit. The refitted model is not regularized.
- **distributed** (*bool*) – If True, the model uses distributed methods for fitting, will raise an error if True and partitions is None.
- **generator** (*function*) – generator used to partition the model, allows for handling of out of memory/parallel computing.
- **partitions** (*scalar*) – The number of partitions desired for the distributed estimation.
- **threshold** (*scalar or array-like*) – The threshold below which coefficients are zeroed out, only used for distributed estimation

Returns

Return type A RegularizedResults instance.

Notes

The elastic net uses a combination of L1 and L2 penalties. The implementation closely follows the glmnet package in R.

The function that is minimized is:

$$0.5 * RSS/n + alpha * ((1 - L1\_wt) * |params|_2^2/2 + L1\_wt * |params|_1)$$

where RSS is the usual regression sum of squares, n is the sample size, and $| * |_1$ and $| * |_2$ are the L1 and L2 norms.

For WLS and GLS, the RSS is calculated using the whitened endog and exog data.

Post-estimation results are based on the same data used to select variables, hence may be subject to overfitting biases.

The elastic_net method uses the following keyword arguments:

maxiter [int] Maximum number of iterations

cnvrg_tol [float] Convergence threshold for line searches

zero_tol [float] Coefficients below this threshold are treated as zero.

The square root lasso approach is a variation of the Lasso that is largely self-tuning (the optimal tuning parameter does not depend on the standard deviation of the regression errors). If the errors are Gaussian, the tuning parameter can be taken to be

alpha = 1.1 * np.sqrt(n) * norm.ppf(1 - 0.05 / (2 * p))

where n is the sample size and p is the number of predictors.

The square root lasso uses the following keyword arguments:

zero_tol [float] Coefficients below this threshold are treated as zero.

References

Friedman, Hastie, Tibshirani (2008). Regularization paths for generalized linear models via coordinate descent. Journal of Statistical Software 33(1), 1-22 Feb 2010.

A Belloni, V Chernozhukov, L Wang (2011). Square-root Lasso: pivotal recovery of sparse signals via conic programming. Biometrika 98(4), 791-806. https://arxiv.org/pdf/1009.5689.pdf

`hessian`(*params*, *scale=None*)

Evaluate the Hessian function at a given point.

Parameters

- **params** (*array-like*) – The parameter vector at which the Hessian is computed.
- **scale** (*float or None*) – If None, return the profile (concentrated) log likelihood (profiled over the scale parameter), else return the log-likelihood using the given scale value.

Returns

Return type The Hessian matrix.

`hessian_factor`(*params*, *scale=None*, *observed=True*)

Weights for calculating Hessian

Parameters

- **params** (*ndarray*) – parameter at which Hessian is evaluated
- **scale** (*None or float*) – If scale is None, then the default scale will be calculated. Default scale is defined by *self.scaletype* and set in fit. If scale is not None, then it is used as a fixed scale.
- **observed** (*bool*) – If True, then the observed Hessian is returned. If false then the expected information matrix is returned.

Returns **hessian_factor** – A 1d weight vector used in the calculation of the Hessian. The hessian is obtained by *(exog.T * hessian_factor).dot(exog)*

Return type ndarray, 1d

`loglike`(*params*, *scale=None*)

The likelihood function for the OLS model.

Parameters

- **params** (*array-like*) – The coefficients with which to estimate the log-likelihood.
- **scale** (*float or None*) – If None, return the profile (concentrated) log likelihood (profiled over the scale parameter), else return the log-likelihood using the given scale value.

Returns

Return type The likelihood function evaluated at params.

`score`(*params*, *scale=None*)

Evaluate the score function at a given point.

The score corresponds to the profile (concentrated) log-likelihood in which the scale parameter has been profiled out.

Parameters

- **params** (*array-like*) – The parameter vector at which the score function is computed.

- **scale** (*float or None*) – If None, return the profile (concentrated) log likelihood (profiled over the scale parameter), else return the log-likelihood using the given scale value.

Returns

Return type The score vector.

whiten(*Y*)
OLS model whitener does nothing: returns Y.

trtools.associaTR.**main**(*args*)

trtools.associaTR.**perform_gwas**(*outfname*, *tr_vcf*, *phenotype_name*, *traits_fnames*, *vcftype*, *same_samples*, *sample_fname*, *region*, *non_major_cutoff*, *beagle_dosages*, *plotting_phenotype_fname*, *paired_genotype_plot*, *plot_phenotype_residuals*, *plotting_ci_alphas*, *imputed_ukb_strs_paper_period_check*)

trtools.associaTR.**perform_gwas_helper**(*outfile*, *all_samples*, *get_genotype_iter*, *phenotype_name*, *trait_fnames*, *same_samples*, *sample_fname*, *beagle_dosages*, *plotting_phenotype_fname*, *paired_genotype_plot*, *plot_phenotype_residuals*, *plotting_ci_alphas*)

trtools.associaTR.**run**()

trtools.compareSTR module

trtools.compareSTR.**CalcR2**(*format_bin_results*)
Calculate the squared (pearson) correlation coefficient for the values in this bin.

Calculation is done using the formulas: n = numcalls var(X) = sum(X_i**2)/n - [sum(X_i)/n]**2 covar(X,Y) = sum(X_i*Y_i)/n - sum(X_i)/n * sum(Y_i)/n r^2 = covar(X,Y)**2/(var(X) * var(Y))

Parameters **format_bin_results** (*Dict[str, int]*) – See the method NewOverallForamtBin

Returns r^2, or np.nan if one of the two vcfs has no variance in this format bin

Return type float

trtools.compareSTR.**GetBubbleLegend**(*coordinate_counts*)
Get three good bubble legend sizes to use

They should be nice round numbers spanning the orders of magnitude of the dataset

Parameters **coordinate_counts** – set of counts for coordinates in the graph

Returns **legend_values** – List of three or fewer representative sample sizes to use for bubble legend

Return type list of int

trtools.compareSTR.**GetFormatFields**(*format_fields*, *format_binsizes*, *format_fileoption*, *vcfreaders*)
Get which FORMAT fields to stratify on

Also perform some checking on user arguments

Parameters

- **format_fields** (*str*) – Comma-separated list of FORMAT fields to stratify on
- **format_binsizes** (*str*) – Comma-separated list of min:max:binsize, one for each FORMAT field.

- **format_fileoption** (*{0, 1, 2}*) – Whether each format field needs to be in both readers (0), reader 1 (1) or reader 2 (2)
- **vcfreaders** (*list of vcf.Reader*) – List of readers. Needed to check if required FORMAT fields are present

Returns

- **formats** (*list of str*) – List of FORMAT fields to stratify on
- **format_bins** (*List[List[float]]*) – List of bin start/stop coords for each FORMAT field

`trtools.compareSTR.`**NewOverallFormatBin**`()`

Return an empty bin for the overall dictionary.

Returns Contains the fields: conc_len_count conc_seq_cont numcalls total_len_1 total_len_2 total_len_11 total_len_12 total_len_22

Return type Dict[str, Union[int, float]]

`trtools.compareSTR.`**NewOverallPeriod**`(`*format_fields*, *format_bins*`)`

Return an empty dictionary containing bins for each format stratification and for 'ALL' (no format stratification).

Returns

Return type The empty dictionary.

`trtools.compareSTR.`**OutputBubblePlot**`(`*bubble_results*, *outprefix*, *minval=None*, *maxval=None*`)`

Output bubble plot of gtsum1 vs. gtsum2

Parameters

- **bubble_results** – counts of sum1 vs sum2
- **outprefix** (*str*) – Prefix to name output file

`trtools.compareSTR.`**OutputLocusMetrics**`(`*locus_results*, *outprefix*, *noplot*`)`

Output per-locus metrics

Outputs text file and plot of per-locus metrics outprefix + "-locuscompare.tab" outprefix + "-locuscompare.pdf"

Parameters

- **locus_results** (*Dict[str, Any]*) – The info needed to write the output file
- **outprefix** (*str*) – Prefix to name output file
- **noplot** (*bool*) – If True, don't output plots

`trtools.compareSTR.`**OutputOverallMetrics**`(`*overall_results*, *format_fields*, *format_bins*, *outprefix*`)`

Output overall accuracy metrics

Output metrics overall, by period, and by FORMAT bins Output results to outprefix+"-overall.tab"

Parameters

- **overall_results** (*Dict[str, Any]*) – Info needed to write the tabfile
- **format_fields** (*List[str]*) – List of FORMAT fields to stratify by
- **format_bins** (*List[List[float]]*) – List of bin start/stop coords for each FORMAT field
- **outprefix** (*str*) – Prefix to name output file

`trtools.compareSTR.`**OutputSampleMetrics**`(`*sample_results*, *sample_names*, *outprefix*, *noplot*`)`

Output per-sample metrics

Outputs text file and plot of per-sample metrics outprefix + "-samplecompare.tab" outprefix + "-samplecompare.pdf"

Parameters

- **sample_results** (*Dict[str, any]*) – The info needed to write the output file
- **sample_names** (*List[str]*) –
- **outprefix** (*str*) – Prefix to name output file
- **noplot** (*bool*) – If True, don't output plots

trtools.compareSTR.**UpdateComparisonResults**(*record1*, *record2*, *sample_idxs*, *ignore_phasing*, *stratify_by_period*, *format_fields*, *format_bins*, *stratify_file*, *overall_results*, *locus_results*, *sample_results*, *bubble_results*)

Extract comparable results from a pair of VCF records

Parameters

- **record1** (*trh.TRRecord*) – First record to compare
- **record2** (*trh.TRRecord*) – Second record to compare
- **sample_idxs** (*list of np.array*) – Two arrays, one for each vcf Each array is a list of indicies so that vcf1.samples[index_array1] == vcf2.samples[index_array2] and that this is the set of shared samples
- **stratify_by_period** (*bool*) – If True, also stratify results by period
- **format_fields** (*list of str*) – List of format fields to extract
- **format_bins** (*List[List[float]]*) – List of bin start/stop coords for each FORMAT field
- **stratify_file** (*{0, 1, 2}*) – Specify whether to apply FORMAT stratification to both files (0), or only (1) or (2)
- **overall_results** (*dict*) – Period and format nested dictionary to update.
- **locus_results** (*dict*) – Locus-stratified results dictionary to update.
- **sample_results** (*dict*) – Sample-stratified results dictionary to update.
- **bubble_results** (*dict*) – dictionary of counts to update

trtools.compareSTR.**check_region**(*contigs1*, *contigs2*, *region_str*)

trtools.compareSTR.**getargs**()

trtools.compareSTR.**handle_overlaps**(*records*, *chrom_indices*, *min_chrom_index*)

This function determines whether (two) records in list are comparable Currently only works with record lists which are two records long

Parameters

- **records** (*List[Optional[trh.TRRecord]]*) – List of TRRecords whose comparability is to be determined. If any of them is None, they are not comparable
- **chrom_indices** (*List[int]*) – List of indices of chromosomes of current records
- **min_chrom_index** (*int*) – Smallest index in chrom_indices. All records should have the same chrom_index, otherwise they are not comparable

Returns **comparable** – Result, that says whether records are comparable

Return type bool

`trtools.compareSTR.`**`main`**(*args*)

`trtools.compareSTR.`**`run`**()

trtools.dumpSTR module

`trtools.dumpSTR.`**`ApplyCallFilters`**(*record*, *call_filters*, *sample_info*, *sample_names*)

Apply call-level filters to a record.

Returns a TRRecord object with the FILTER (or DUMPSTR_FILTER) format field updated for each sample. Also updates sample_info with sample level stats

Parameters

- **`record`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record to apply filters to. Note: once this method has been run, this object will be in an inconsistent state. All further use should be directed towards the returned TRRecord object.
- **`call_filters`** (*`List[`*`trtools.dumpSTR.filters.FilterBase`*`]`*) – List of call filters to apply
- **`sample_info`** (*`Dict[str, numpy.ndarray]`*) – Dictionary of sample stats to keep updated, from name of filter to array of length nsamples which counts the number of times that filter has been applied to each sample across all loci
- **`sample_names`** (*`List[str]`*) – Names of all the samples in the vcf. Used for formatting error messages.

Returns A reference to the same underlying cyvcf2.Variant object, which has now been modified to contain all the new call-level filters.

Return type trh.TRRecord

`trtools.dumpSTR.`**`ApplyLocusFilters`**(*record*, *locus_filters*, *loc_info*, *drop_filtered*)

Apply locus-level filters to a record.

If not drop_filtered, then the input record's FILTER field is set as either PASS or the names of the filters which filtered it.

Parameters

- **`record`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record to apply filters to.
- **`call_filters`** – List of locus filters to apply
- **`loc_info`** (*`Dict[str, int]`*) – Dictionary of locus stats to keep updated, from name of filter to count of times the filter has been applied
- **`drop_filtered`** (*`bool`*) – Whether or not filtered loci should be written to or dropped from the output vcf.
- **`locus_filters`** (*`List[`*`trtools.dumpSTR.filters.FilterBase`*`]`*) –

Returns **locus_filtered** – True if this locus was filtered

Return type bool

`trtools.dumpSTR.`**`BuildCallFilters`**(*args*)

Build list of locus-level filters to include

Parameters **`args`** (*`argparse namespace`*) – User input arguments used to decide on filters

Returns **filter_list** – List of call-level filters to apply

Return type list of filters.Filter

`trtools.dumpSTR.`**`BuildLocusFilters`**`(`*args*`)`

Build list of locus-level filters to include.

These filters should in general not be tool specific

Parameters **`args`** (*`argparse namespace`*) – User input arguments used to decide on filters

Returns **filter_list** – List of locus-level filters

Return type list of filters.Filter

`trtools.dumpSTR.`**`CheckAdVNTRFilters`**`(`*format_fields*, *args*`)`

Check adVNTR call-level filters

Parameters

- **`format_fields`** – The format fields used in this VCF
- **`args`** (*`argparse namespace`*) – Contains user arguments

Returns **checks** – Set to True if all filters look ok. Set to False if filters are invalid

Return type bool

`trtools.dumpSTR.`**`CheckEHFilters`**`(`*format_fields*, *args*`)`

Check ExpansionHunter call-level filters

Parameters

- **`format_fields`** – The format fields used in this VCF
- **`args`** (*`argparse namespace`*) – Contains user arguments

Returns **checks** – Set to True if all filters look ok. Set to False if filters are invalid

Return type bool

`trtools.dumpSTR.`**`CheckFilters`**`(`*format_fields*, *args*, *vcftype*, *is_beagle*`)`

Perform checks on user input for filters.

Assert that user input matches the type of the input vcf.

Parameters

- **`format_fields`** (*`Set[str]`*) – The format fields used in this VCF
- **`args`** (*`argparse.Namespace`*) – Contains user arguments
- **`vcftype`** (`trtools.utils.tr_harmonizer.VcfTypes`) – Specifies which tool this VCF came from.
- **`is_beagle`** (*`bool`*) – Was this VCF generated by Beagle imputation?

Returns **checks** – Set to True if all filters look ok. Set to False if filters are invalid

Return type bool

`trtools.dumpSTR.`**`CheckGangSTRFilters`**`(`*format_fields*, *args*`)`

Check GangSTR call-level filters

Parameters

- **`format_fields`** – The format fields used in this VCF
- **`args`** (*`argparse namespace`*) – Contains user arguments

Returns **checks** – Set to True if all filters look ok. Set to False if filters are invalid

Return type bool

`trtools.dumpSTR.`**`CheckHipSTRFilters`**`(`*format_fields*, *args*`)`

Check HipSTR call-level filters

Parameters

- **`format_fields`** – The format fields used in this VCF
- **`args`** (*`argparse namespace`*) – Contains user arguments

Returns **checks** – Set to True if all filters look ok. Set to False if filters are invalid

Return type bool

`trtools.dumpSTR.`**`CheckLocusFilters`**`(`*args*, *vcftype*, *is_beagle*`)`

Perform checks on user inputs for locus-level filters

Parameters

- **`args`** (*`argparse namespace`*) – Contains user arguments
- **`vcftype`** (*`enum.`*) – Specifies which tool this VCF came from. Must be included in trh.VCFTYPES
- **`is_beagle`** (*`bool`*) – Was this VCF generated by Beagle imputation?

Returns **checks** – Set to True if all filters look ok. Set to False if filters are invalid

Return type bool

`trtools.dumpSTR.`**`CheckPopSTRFilters`**`(`*format_fields*, *args*`)`

Check PopSTR call-level filters

Parameters

- **`format_fields`** – The format fields used in this VCF
- **`args`** (*`argparse namespace`*) – Contains user arguments

Returns **checks** – Set to True if all filters look ok. Set to False if filters are invalid

Return type bool

`trtools.dumpSTR.`**`GetAllCallFilters`**`(`*call_filters*`)`

List all possible call filters

Parameters **`call_filters`** (*`list of filters.Reason`*) – List of all call-level filters

Returns **reasons** – A list of call-level filter reason strings

Return type list of str

`trtools.dumpSTR.`**`MakeWriter`**`(`*outfile*, *invcf*, *command*`)`

Create a VCF writer with a dumpSTR header

Adds a header line with the dumpSTR command used

Parameters

- **`outfile`** (*`str`*) – Name of the output file
- **`invcf`** (*`vcf.Reader object`*) – Input VCF. Used to grab header info
- **`command`** (*`str`*) – String command used to run dumpSTR

Returns **writer** – VCF writer initialized with header of input VCF Set to None if we had a problem writing the file

Return type vcf.Writer object

`trtools.dumpSTR.`**`WriteLocLog`**(*loc_info*, *fname*)

Write locus-level features to log file

Parameters

- **loc_info** (*dict of str->value*) – Dictionary containing locus-level stats. Must have at least keys: 'totalcalls', 'PASS'
- **fname** (*str*) – Output log filename

Returns **success** – Set to true if outputting the log was successful

Return type bool

`trtools.dumpSTR.`**`WriteSampLog`**(*sample_info*, *sample_names*, *fname*)

Write sample-level features to log file.

Parameters

- **sample_info** (*Dict[str, numpy.ndarray]*) – Mapping from statistic name to 1D array of values per sample
- **sample_names** (*List[str]*) – List of sample names, same length as above arrays
- **fname** (*str*) – Output filename

`trtools.dumpSTR.`**`getargs`**()

`trtools.dumpSTR.`**`main`**(*args*)

`trtools.dumpSTR.`**`run`**()

trtools.dumpSTR.filters module

Locus-level and Call-level VCF filters

class `trtools.dumpSTR.filters.`**`CallFilterMaxValue`**(*name*, *field*, *threshold*)

Bases: *trtools.dumpSTR.filters.Reason*

Generic call-level filter based on maximum allowed value for a field.

Extends Reason class. For any call-level value, such as DP, this class can be used to make a filter based on the maximum allowed value for that field.

Parameters

- **name** (*str*) – The name of the filter to put in the FORMAT:FILTER field of filtered calls.
- **field** (*str*) – The FORMAT field to filter on
- **threshold** (*float*) – The maximum allowed value for the field.

`name`

The name of the filter to put in the FORMAT:FILTER field of filtered calls.

Type str

`field`

The FORMAT field to filter on

Type str

`threshold`

The maximum allowed value for the field.

Type float

Examples

```
>>> max_dp_filt = CallFilterMaxValue("HIGHDP","DP",1000)
```

class trtools.dumpSTR.filters.**CallFilterMinValue**(*name*, *field*, *threshold*)

Bases: *trtools.dumpSTR.filters.Reason*

Generic call-level filter based on minimum allowed value for a field.

Extends Reason class. For any call-level value, such as DP, this class can be used to make a filter based on the minimum allowed value for that field.

Parameters

- **name** (*str*) – The name of the filter to put in the FORMAT:FILTER field of filtered calls.
- **field** (*str*) – The FORMAT field to filter on
- **threshold** (*float*) – The minimum allowed value for the field.

name

The name of the filter to put in the FORMAT:FILTER field of filtered calls.

Type str

field

The FORMAT field to filter on

Type str

threshold

The minimum allowed value for the field.

Type float

Examples

```
>>> min_dp_filt = CallFilterMinValue("LOWDP","DP",10)
```

class trtools.dumpSTR.filters.**FilterBase**

Bases: object

Base class for locus level filters. Just defines the interface

description()

filter_name()

name = **'NotYetImplemented'**

class trtools.dumpSTR.filters.**Filter_LocusHrun**

Bases: *trtools.dumpSTR.filters.FilterBase*

Class to filter VCF records for penta- or hexanucleotide STRs with long homopolymer runs

This only works on HipSTR VCFs. STRs with long homopolymer runs have been shown to be difficult for HipSTR to call. This filter removes 5-mers with homopolymer runs >= len 5 and 6-mers with homopolymer runs >= len 6

filter_name()

name = 'HRUN'
The name of the filter

class trtools.dumpSTR.filters.**Filter_MaxLocusHet**(*max_locus_het*, *uselength=False*)
Bases: *trtools.dumpSTR.filters.FilterBase*

Class to filter VCF records by maximum heterozygosity

This class extends Base

Parameters

- **max_locus_het** (*float*) – Filters calls with heterozygosity greater than this
- **vcftype** (*trh.VCFTYPES*) – the type of the VCF we're working with
- **uselength** (*bool, optional*) – If set to true, consider all alleles with the same length as the same

threshold
Filters calls with heterozygosity greater than this

Type float

vcftype
the type of the VCF we're working with

Type trh.VCFTYPES

uselength
If set to true, consider all alleles with the same length as the same

Type bool, optional

filter_name()

name = 'HETHIGH'
The name of the filter

class trtools.dumpSTR.filters.**Filter_MinLocusCallrate**(*min_locus_callrate*)
Bases: *trtools.dumpSTR.filters.FilterBase*

Class to filter VCF records by call rate

This class extends Base

Parameters **min_locus_callrate** (*float*) – Filters calls with lower than this fraction called

threshold
Filters calls with lower than this fraction called Derived from the input min_locus_callrate

Type float

filter_name()

name = 'CALLRATE'
The name of the filter

class trtools.dumpSTR.filters.**Filter_MinLocusHWEP**(*min_locus_hwep*, *uselength=False*)
Bases: *trtools.dumpSTR.filters.FilterBase*

Class to filter VCF records by minimum HWE p-value

This class extends Base

Parameters

- **min_locus_hwep** (*float*) – Filters calls with HWE p-value lower than this

- **vcftype** (*trh.VCFTYPES*) – the type of the VCF we're working with
- **uselength** (*bool, optional*) – If set to true, consider all alleles with the same length as the same

threshold

Filters calls with HWE p-value lower than this

Type float

vcftype

the type of the VCF we're working with

Type trh.VCFTYPES

uselength

If set to true, consider all alleles with the same length as the same

Type bool, optional

filter_name()

name = 'HWE'

The name of the filter

class trtools.dumpSTR.filters.**Filter_MinLocusHet**(*min_locus_het*, *uselength=False*)

Bases: *trtools.dumpSTR.filters.FilterBase*

Class to filter VCF records by minimum heterozygosity

This class extends Base

Parameters

- **min_locus_het** (*float*) – Filters calls with heterozygosity lower than this
- **vcftype** (*trh.VCFTYPES*) – the type of the VCF we're working with
- **uselength** (*bool, optional*) – If set to true, consider all alleles with the same length as the same

threshold

Filters calls with heterozygosity lower than this

Type float

vcftype

the type of the VCF we're working with

Type trh.VCFTYPES

uselength

If set to true, consider all alleles with the same length as the same

Type bool, optional

filter_name()

name = 'HETLOW'

The name of the filter

class trtools.dumpSTR.filters.**GangSTRCallBadCI**

Bases: *trtools.dumpSTR.filters.Reason*

Filter GangSTR calls where the ML genotype estimate is outside of CI.

If 95% confidence interval does not include the maximum likelihood genotype call, the call is filtered.

Extends Reason class. Based on REPCI and REPCN fields.

name = 'GangSTRCallBadCI'
The name of the filter

class trtools.dumpSTR.filters.**GangSTRCallExpansionProbHet**(*threshold*)
Bases: *trtools.dumpSTR.filters.Reason*

Filter GangSTR calls with low probability for heterozygous expansion.

Extends Reason class. Based on the QEXP field. Filter if QEXP[:, 1] (prob het expansion above INFO:THRESHOLD) is less than the threshold.

Parameters **threshold** (*float*) – Minimum heterozygous expansion probability

threshold
Minimum heterozygous expansion probability

Type float

name = 'GangSTRCallExpansionProbHet'
The name of the filter

class trtools.dumpSTR.filters.**GangSTRCallExpansionProbHom**(*threshold*)
Bases: *trtools.dumpSTR.filters.Reason*

Filter GangSTR calls with low probability for homozygous expansion.

Extends Reason class. Based on the QEXP field. Filter if QEXP[:, 2] (prob hom expansion above INFO:THRESHOLD) is less than the threshold.

Parameters **threshold** (*float*) – Minimum homozygous expansion probability

threshold
Minimum homozygous expansion probability

Type float

name = 'GangSTRCallExpansionProbHom'
The name of the filter

class trtools.dumpSTR.filters.**GangSTRCallExpansionProbTotal**(*threshold*)
Bases: *trtools.dumpSTR.filters.Reason*

Filter GangSTR calls with low probability for expansion (heterozygous or homozygous)

Extends Reason class. Based on the QEXP field. Filter if QEXP[1]+QEXP[2] (prob het or hom expansion above INFO:THRESHOLD) is less than the threshold.

Parameters **threshold** (*float*) – Minimum expansion probability

threshold
Minimum expansion probability

Type float

name = 'GangSTRCallExpansionProbTotal'
The name of the filter

class trtools.dumpSTR.filters.**GangSTRCallSpanBoundOnly**
Bases: *trtools.dumpSTR.filters.Reason*

Filter GangSTR calls where only spanning or flanking reads were identified.

Extends Reason class. Based on RC field.

name = 'GangSTRCallSpanBoundOnly'
The name of the filter

class trtools.dumpSTR.filters.**GangSTRCallSpanOnly**
Bases: *trtools.dumpSTR.filters.Reason*

Filter GangSTR calls where only spanning reads were identified.

Extends Reason class. Based on RC field.

name = 'GangSTRCallSpanOnly'
FILTER field of filtered calls.

Type The name of the filter to put in the FORMAT

class trtools.dumpSTR.filters.**HipSTRCallFlankIndels**(*threshold*)
Bases: *trtools.dumpSTR.filters.Reason*

Filter HipSTR calls with many indels in flanks

Extends Reason class. Filters on the percentage of reads with indels in flanks. Based on FORMAT:DFLANKINDEL and FORMAT:DP fields.

Parameters **threshold** (*float*) – Minimum percent of reads that can have indels in their flanks

threshold
Minimum percent of reads that can have indels in their flanks

Type float

name = 'HipSTRCallFlankIndels'
The name of the filter

class trtools.dumpSTR.filters.**HipSTRCallMinSuppReads**(*threshold*)
Bases: *trtools.dumpSTR.filters.Reason*

Filter HipSTR calls for which alleles are supported by too few reads

Extends Reason class. Filters on the number of reads supporting each called allele. Based on FORMAT:ALLREADS and FORMAT:GB fields. Assumes that number of supporting reads is zero if: * that read length is not present in ALLREADS * or ALLREADS is unset for that sample ('.') * or ALLREADS is not present at that locus

Parameters **threshold** (*int*) – Minimum number of reads supporting each allele

threshold
Minimum number of reads supporting each allele

Type int

name = 'HipSTRMinSuppReads'
The name of the filter

class trtools.dumpSTR.filters.**HipSTRCallStutter**(*threshold*)
Bases: *trtools.dumpSTR.filters.Reason*

Filter HipSTR calls with many stutter reads

Extends Reason class. Filters on the percentage of reads with stutter errors Based on FORMAT:DSTUTTER and FORMAT:DP fields.

Parameters **threshold** (*float*) – Minimum percent of reads that can have stutter errors

threshold
Minimum percent of reads that can have stutter errors

Type float

`name = 'HipSTRCallStutter'`
The name of the filter

`class trtools.dumpSTR.filters.PopSTRCallRequireSupport(threshold)`
Bases: *trtools.dumpSTR.filters.Reason*

Filter PopSTR calls not supported by enough reads

Extends Reason class. Relies on FORMAT:AD field.

Parameters **threshold** (*int*) – Require this many reads supporting each called allele.

threshold
Require this many reads supporting each called allele.

Type int

`name = 'PopSTRCallRequireSupport'`
The name of the filter

`class trtools.dumpSTR.filters.Reason`
Bases: `object`

Base call-level filter class.

Other classes extend this for each different call-level filter. Classes that extend this must implement a __call__ function that gets applied to each call. The __call__ function returns a 1D array of values, one per sample. For numeric arrays, nan values indicate the sample wasn't filtered, and any other value indicates that the sample was filtered (where the value indicates why).

`GetReason()`

`name = ''`
FILTER field of filtered calls.

Type The name of the filter to put in the FORMAT

`trtools.dumpSTR.filters.create_region_filter(name, filename)`
Creates a locus-level filter based on a file of regions.

Builds and returns a class extending Base that can be used to filter any records overlapping intervals in the input BED file

Parameters

- **name** (*str*) – Name of the region filter to create. This will go in the FILTER field of the output VCF
- **filename** (*str*) – BED file containing the regions. Must be sorted by chrom, start If it's not bgzipped and tabixed, we'll attempt to do that.
- **Returns** –
- **filter_regions** (*Base object.*) – Returns None if we fail to load the regions
- **-------** –

trtools.mergeSTR module

trtools.mergeSTR.**GetAltAlleles**(*ref_allele*, *current_records*, *mergelist*, *vcftype*)

Get list of alt alleles

Parameters

- **ref_allele** (*str*) – The (flank trimmed) reference allele, in upper case
- **current_records** (*list of vcf.Record*) – List of records being merged
- **mergelist** (*list of bool*) – Indicates whether each record is included in merge
- **vcftype** (*Union[str, trtools.utils.tr_harmonizer.VcfTypes]*) – The type of the VCFs these records came from

Returns

(alts, mappings) – alts is a list of alternate allele strings in all uppercase. mappings is a list of length equal to the number of records being merged. For record n, mappings[n] is a numpy array where an allele with index i in the original record has an index of mappings[n][i] in the output merged record. (the indicies stored in the arrays are number strings for fast output, e.g. '1' or '2') For example if the output record has ref allele 'A' and alternate alleles 'AA,AAA,AAAA' and input record n has ref allele 'A' and alternate alleles 'AAAA,AAA' then mappings[n] would be np.array(['0', '3', '2']).

```
original index      new index
rec_n.alleles[0] == out_rec.alleles[0] == 'A'
rec_n.alleles[1] == out_rec.alleles[3] == 'AAAA'
rec_n.alleles[2] == out_rec.alleles[2] == 'AAA'
```

Return type (list of str, list of np.ndarray)

trtools.mergeSTR.**GetAltsByKey**(*current_records*, *mergelist*, *key*)

Parameters

- **current_records** (*List[trtools.utils.tr_harmonizer.TRRecord]*) –
- **mergelist** (*List[bool]*) –
- **key** (*Any*) –

trtools.mergeSTR.**GetID**(*idval*)

Get the ID for a a record

If not set, output "."

Parameters **idval** (*str*) – ID of the record

Returns **idval** – Return ID. if None, return "."

Return type str

trtools.mergeSTR.**GetInfoItem**(*current_records*, *mergelist*, *info_field*, *fail=True*)

Get INFO item for a group of records

Make sure it's the same across merged records if fail=True, die if items not the same. if fail=False, only do something if we have a rule on how to handle that field

Parameters

- **current_records** (*list of vcf.Record*) – List of records being merged

- **mergelist** (*list of bool*) – List of indicators of whether to merge each record
- **info_field** (*str*) – INFO field being merged
- **fail** (*bool*) – If True, throw error if fields don't have same value

Returns **infostring** – INFO string to add (key=value)

Return type str

`trtools.mergeSTR.`**GetRefAllele**(*current_records*, *mergelist*, *vcfType*)

Get reference allele for a set of records

Parameters

- **current_records** (*list of vcf.Record*) – List of records being merged
- **mergelist** (*list of bool*) – Indicates whether each record is included in merge
- **vcfType** (`trtools.utils.tr_harmonizer.VcfTypes`) –

Returns **ref** – Reference allele string. Set to None if conflicting references are found.

Return type str

`trtools.mergeSTR.`**HarmonizeIfNotNone**(*records*, *vcf_type*)

Parameters

- **records** (*List[Optional[*`trtools.utils.tr_harmonizer.TRRecord`*]]*) –
- **vcf_type** (`trtools.utils.tr_harmonizer.VcfTypes`) –

`trtools.mergeSTR.`**MergeRecords**(*readers*, *vcftype*, *num_samples*, *current_records*, *mergelist*, *vcfw*, *useinfo*, *useformat*, *format_type*)

Merge records from different files

Merge all records with indicator set to True in mergelist Output merged record to vcfw If the merged records were created by HipSTR and contain flanking BPs, these BPs are removed

Parameters

- **readers** (*list of vcf.Reader*) – List of readers being merged
- **vcftype** (*Union[str,* `trtools.utils.tr_harmonizer.VcfTypes`*]*) – Type of the readers
- **num_samples** (*list of int*) – Number of samples per vcf
- **current_records** (*list of vcf.Record*) – List of current records for each reader
- **mergelist** (*list of bool*) – Indicates whether to include each reader in merge
- **vcfw** (*file*) – File to write output to
- **useinfo** (*list of (str, bool)*) – List of (info field, required) to use downstream
- **useformat** (*list of str*) – List of format field strings to use downstream
- **format_type** (*list of String*) – The type of each format field

Return type None

`class trtools.mergeSTR.`**TextIO**(**args*, ***kwds*)

Bases: `IO[str]`

Typed version of the return of open() in text mode.

abstract property buffer: BinaryIO

abstract property encoding: str

abstract property errors: Optional[str]

abstract property line_buffering: bool

abstract property newlines: Any

trtools.mergeSTR.**WriteMergedHeader**(*vcfw*, *args*, *readers*, *cmd*, *vcftype*)

Write merged header for VCFs in args.vcfs

Also do some checks on the VCFs to make sure merging is appropriate. Return info and format fields to use

Parameters

- **vcfw** (*file object*) – Writer to write the merged VCF
- **args** (*argparse namespace*) – Contains user options
- **readers** (*list of vcf.Reader*) – List of readers to merge
- **cmd** (*str*) – Command used to call this program
- **vcftype** (*str*) – Type of VCF files being merged

Returns

- **useinfo** (*list of (str, bool)*) – List of (info field, required) to use downstream
- **useformat** (*list of str*) – List of format field strings to use downstream

Return type Union[Tuple[List[Tuple[str, bool]], List[str]], Tuple[None, None]]

trtools.mergeSTR.**WriteSampleData**(*vcfw*, *record*, *alleles*, *formats*, *format_type*, *mapping*)

Output sample FORMAT data

Writes a string representation of the GT and other format fields for each sample in the record, with tabs in between records

Parameters

- **vcfw** (*file*) – File to write output to
- **record** (*cyvcf2.Varaint*) – VCF record being summarized
- **alleles** (*list of str*) – List of REF + ALT alleles
- **formats** (*list of str*) – List of VCF FORMAT items
- **format_type** (*list of String*) – The type of each format field
- **mapping** (*np.ndarray*) – See GetAltAlleles

Return type None

trtools.mergeSTR.**getargs**()

Return type Any

trtools.mergeSTR.**main**(*args*)

Parameters **args** (*Any*) –

Return type int

`trtools.mergeSTR.`**`run`**`()`

Return type None

trtools.prancSTR module

`trtools.prancSTR.`**`ComputePvalue`**`(`*reads*, *A*, *B*, *best_C*, *best_f*, *stutter_probs*`)`

Compute pvalue testing H0:f=0

Parameters

- **reads** (*list of int*) – list of repeat lengths seen in each read
- **A** (*int*) – First allele of the genotype
- **B** (*int*) – Second allele of the genotype
- **best_C** (*integer*) – Estimated mosaic allele
- **best_f** (*float*) – mosaic fraction
- **stutter_probs** (*list of floats*) – stutter probs for each delta

Returns **pval** – P-value testing H0: f=0

Return type float

`trtools.prancSTR.`**`ConfineRange`**`(`*x*, *minval*, *maxval*`)`

Confine the range of a nmber to lie between minval and maxval

Parameters

- **x** (*numeric*) – The value to be constrained
- **minval** (*numeric*) – The minimum value the output can take
- **maxval** (*numeric*) – The maximum value the output can take

Returns **x_cons** – New value, which cannot exceed maxval or go below minval

Return type numeric

`trtools.prancSTR.`**`ExtractAB`**`(`*trrecord*`)`

Extract list of <A,B> for each sample

Parameters **trrecord** (*trh.TRRecord*) – TRRecord object for the locus

Returns **genotypes** – [(A,B), ..] genotypes for each sample given in terms of bp diff from ref

Return type list of list of ints

`trtools.prancSTR.`**`ExtractReadVector`**`(`*mallreads*, *period*`)`

Extract reads vector from MALLREADS, MALLREADS has format: allele1|readcount1;allele2|readcount2

Parameters

- **mallreads** (*str*) – MALLREADS string from HipSTR output
- **period** (*int*) – STR unit length

Returns **reads** – List with one entry per read. Given in terms of difference in repeats from reference

Return type list of int

`trtools.prancSTR.`**`Just_C_Pred`**`(`*reads*, *A*, *B*, *f*, *stutter_probs*`)`

Predict C, holding f constant

Parameters

- **reads** (*list of int*) – list of repeat lengths seen in each read
- **A** (*int*) – First allele of the genotype
- **B** (*int*) – Second allele of the genotype
- **f** (*float*) – Mosaic fraction
- **stutter_probs** (*list of floats*) – stutter probs for each delta

Returns **C** – mosaic allele

Return type int

`trtools.prancSTR.`**`Just_F_Pred`**(*reads*, *A*, *B*, *C*, *stutter_probs*)

Predict f, holding C constant

Parameters

- **reads** (*list of int*) – list of repeat lengths seen in each read
- **A** (*int*) – First allele of the genotype
- **B** (*int*) – Second allele of the genotype
- **C** (*integer*) – Mosaic allele
- **stutter_probs** (*list of floats*) – stutter probs for each delta

Returns **f** – mosaic fraction

Return type float

`trtools.prancSTR.`**`Likelihood_mosaic`**(*A*, *B*, *C*, *f*, *reads*, *stutter_probs*)

Compute likelihood of observing the reads, given true genotype=A,B and mosaic allele C, mosaic fraction f

Parameters

- **reads** (*list of int*) – list of repeat lengths seen in each read
- **A** (*int*) – First allele of the genotype
- **B** (*int*) – Second allele of the genotype
- **C** (*integer*) – Mosaic allele
- **stutter_probs** (*list of floats*) – stutter probs for each delta
- **f** (*float*) – mosaic fraction

Returns **sum_likelihood** – sum of max likelihood calculated for each read

Return type float

`trtools.prancSTR.`**`MaximizeMosaicLikelihoodBoth`**(*reads*, *A*, *B*, *stutter_probs*, *maxiter=100*, *locname='None'*, *quiet=False*)

Find the maximum likelihood values of C: mosaic allele f: mosaic fraction

Parameters

- **reads** (*list of int*) – list of repeat lengths seen in each read
- **A** (*int*) – First allele of the genotype
- **B** (*int*) – Second allele of the genotype
- **stutter_probs** (*list of floats*) – stutter probs for each delta

- **max_iter** (*int (optional)*) – Maximum number of iterations to run the estimation procedure. Default=100
- **locname** (*str (optional)*) – String identifier of the locus. For warning message purposes. Default: "None"
- **quiet** (*bool*) – Don't print out any messages

Returns

- **C** (*int*) – Estimated mosaic allele
- **f** (*float*) – Estimated mosaic fraction

trtools.prancSTR.**SF**(*x*)

Survival function of a point mass at 0

Parameters **x** (*float*) – Observed value

Returns **sf** – Survival function result

Return type float

trtools.prancSTR.**StutterProb**(*delta*, *stutter_u*, *stutter_d*, *stutter_rho*)

Compute P(r_i | genotype; error model)

Parameters

- **delta** (*int*) – Difference in repeat length between observed and underlying allele, given in copy number (r_i-genotype)
- **stutter_u** (*float*) – Probability to see an expansion stutter error
- **stutter_d** (*float*) – Probability to see a deletion stutter error
- **stutter_rho** (*float*) – Step size parameter

Returns **prob** – P(r_i|genotype)

Return type float

trtools.prancSTR.**getargs**()

trtools.prancSTR.**main**(*args*)

trtools.prancSTR.**run**()

trtools.qcSTR module

trtools.qcSTR.**OutputChromCallrate**(*chrom_calls*, *fname*)

Plot number of calls per chromosome

Parameters

- **chrom_calls** (*dict of str->int*) – Number of calls for each chromosome
- **fname** (*str*) – Filename of output plot

trtools.qcSTR.**OutputDiffRefBias**(*diffs_from_ref*, *reflens*, *fname*, *xlim=(0, 100)*, *mingts=100*, *metric='mean'*, *binsize=5*)

Plot reflen vs. mean difference from ref bias plot

Parameters

- **diffs_from_ref** (*list of int*) – Difference of each allele call from the ref allele (in bp)
- **reflens** (*list of int*) – List of reference allele lengths for each call (in bp)

- **fname** (*str*) – Filename of output plot
- **xlim** (*tuple of int, optional*) – Specify the minimum and maximum x-axis range (in bp)
- **mingts** (*int, optional*) – Don't plot data points computed based on fewer than this many genotypes
- **metric** (*str, optional*) – Which metric to plot on the y-axis value. Must be mean or median
- **binsize** (*int, optional*) – Size (in bp) of bins on the x-axis.

trtools.qcSTR.**OutputDiffRefHistogram**(*diffs_from_ref*, *fname*)

Plot histogram of difference in bp from reference allele

Parameters

- **diffs_from_ref** (*list of int*) – Difference of each allele call from the ref allele (in units)
- **fname** (*str*) – Filename of output plot

trtools.qcSTR.**OutputQualityLocusStrat**(*per_call_data*, *loci*, *fname*)

Plot quality of calls, one line for each locus

Parameters

- **per_call_data** (*numpy.ndarray*) – 2D array of qualities of calls where each row is a locus and each col is a sample.
- **loci** (*List[str]*) – List of the IDs of loci
- **fname** (*str*) – Location to save the output plot

trtools.qcSTR.**OutputQualityPerCall**(*per_call_data*, *fname*)

Plot quality of calls as one distribution, irrespective of which sample or locus they came from.

Parameters

- **per_call_data** (*numpy.ndarray*) – 1D array of the qualities of all calls
- **fname** (*str*) – Location to save the output plot

trtools.qcSTR.**OutputQualityPerLocus**(*per_locus_data*, *fname*)

Plot quality of calls per locus

Parameters

- **per_locus_data** (*numpy.ndarray*) – 1D array of an average quality for each locus, defined as the average of qualities of calls across all samples at that locus
- **fname** (*str*) – Location to save the output plot

trtools.qcSTR.**OutputQualityPerSample**(*per_sample_data*, *fname*)

Plot quality of calls per sample

Parameters

- **per_sample_data** (*numpy.ndarray*) – 1D array of the average quality for each sample, defined as the average of qualities of calls across all loci at that sample
- **fname** (*str*) – Location to save the output plot

trtools.qcSTR.**OutputQualitySampleStrat**(*per_call_data*, *samples*, *fname*)

Plot quality of calls, one line for each sample

Parameters

- `per_call_data` (*numpy.ndarray*) – 2D array of qualities of calls where each row is a locus and each col is a sample.
- `samples` (*List[str]*) – List of the names of samples
- `fname` (*str*) – Location to save the output plot

`trtools.qcSTR.`**`OutputSampleCallrate`**(*sample_calls*, *samples*, *fname*)

Plot number of calls per sample

Parameters

- `sample_calls` (*numpy.ndarray*) – 1D array, number of calls for each sample
- `samples` (*List[str]*) – List of names of samples, same len as sample_calls
- `fname` (*str*) – Filename of output plot

`trtools.qcSTR.`**`getargs`**()

`trtools.qcSTR.`**`main`**(*args*)

`trtools.qcSTR.`**`run`**()

trtools.simTR module

`trtools.simTR.`**`CreateAlleleFasta`**(*newseq*, *delta*, *tmpdir*)

Create fasta file for this allele Return the path to the fasta

Parameters

- `newseq` (*str*) – New repeat allele sequence
- `delta` (*int*) – Change in repeat units compared to ref
- `tmpdir` (*str*) – Path to create the fasta in

Returns **fname** – Path to created fasta file

Return type str

`trtools.simTR.`**`GetAlleleSeq`**(*seq_preflank*, *seq_postflank*, *seq_repeat*, *repeat_unit*, *delta*)

Generate a new allele with a change of delta repeat units

Parameters

- `seq_preflank` (*str*) – Sequence upstream of the STR
- `seq_postflank` (*str*) – Sequence downstream of the STR
- `seq_repeat` (*str*) – Sequence of the STR region
- `repeat_unit` (*str*) – Repeat unit sequence
- `delta` (*int*) – Change in repeat units compared to ref
- `tmpdir` (*str*) – Path to create the fasta in

Returns **newseq** – New repeat allele sequence Return None if there was a problem

Return type str

`trtools.simTR.`**`GetMaxDelta`**(*sprob*, *rho*, *pthresh*)

Compute the max delta for which the frequency would be great than pthresh

based on freq = sprob*rho*(1-rho)**(delta-1)

Parameters

- `sprob` (*float*) – Stutter probability
- `rho` (*float*) – Stutter step size parameters
- `pthresh` (*float*) – Minimum frequency threshold

Returns **delta** – Highest delta for which freq>prob Return 0 if no such delta exists, which can happen e.g. with low rho

Return type int

`trtools.simTR.GetTempDir(`*debug=False*, *dir=None*`)`

Create a temporary directory to store intermediate fastas and fastqs

Parameters

- `debug` (*bool*) – Ignored for now
- `dir` (*str*) – Directory in which to create the temporary directory

Returns **dirname** – Path to the temporary directory Return None if there was a problem creating the directory

Return type str

`trtools.simTR.ParseCoordinates(`*coords*`)`

Extract chrom, start, end from coordinate string

Parameters `coords` (*str*) – Coordinate string in the form chrom:start-end

Returns

- **chrom** (*str*) – Chromosome name
- **start** (*int*) – start coordinate
- **end** (*int*) – end coordinate
- *If we encounter an error parsing, then*
- *chrom, start, end are None*

`trtools.simTR.SimulateReads(`*newfasta*, *coverage*, *read_length*, *single*, *insert*, *sd*, *tmpdir*, *delta*, *art_cmd*`)`

Run ART on our dummy fasta file with specified parameters

Parameters

- `newfasta` (*str*) – Path to dummy fasta file
- `coverage` (*int*) – Desired coverage level (ART -f)
- `read_length` (*int*) – Read length (ART -l)
- `single` (*bool*) – Use single-end read mode
- `insert` (*float*) – Mean fragment length (ART -m)
- `sd` (*float*) – Std dev of fragment length distribution (ART -s)
- `tmpdir` (*str*) – Path to create the fasta in
- `delta` (*int*) – Difference in repeat units from reference Used for naming files
- `art_cmd` (*str*) – Command to run ART

Returns **fq1file, fq2file** – Paths to fastq file output for the two read pairs. Return None, None if failed If single end mode, fq2file is None

Return type str, str

`trtools.simTR.`**`WriteCombinedFastqs`**`(`*fqfiles*, *fname*`)`

Concatenate fastq files to output

Parameters

- **`fqfiles`** (*`list of str`*) – List of paths to fastqfiles to concatenate
- **`fname`** (*`str`*) – Name of final output file

`trtools.simTR.`**`getargs`**`()`

`trtools.simTR.`**`main`**`(`*args*`)`

`trtools.simTR.`**`run`**`()`

trtools.statSTR module

`trtools.statSTR.`**`GetAFreq`**`(`*trrecord*, *sample_indexes=[None]*, *count=False*, *uselength=True*`)`

Return allele frequency for a TR

Parameters

- **`trrecord`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record that we are computing the statistic for
- **`sample_indexes`** (*`List[Any]`*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **`count`** (*`bool`*) – If True, return allele counts rather than allele frequencies
- **`uselength`** (*`bool`*) – Whether we should collapse alleles by length

Returns **allele_freqs_strs** – Format: allele1:freq1,allele2:freq2,etc. for each sample group Only alleles with more than one call in a group are reported for that group. Groups with no called alleles are reported as '.'

Return type list of str

`trtools.statSTR.`**`GetEntropy`**`(`*trrecord*, *sample_indexes=[None]*, *uselength=True*`)`

Compute the entropy of a locus

This is the (bit) entropy of the distribution of alleles called at that locus. See wikipedia for the definition of entropy.

Parameters

- **`trrecord`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record that we are computing the statistic for
- **`sample_indexes`** (*`List[Any]`*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **`uselength`** (*`bool`*) – Whether we should collapse alleles by length

Returns **heterozygosity** – For each sample list, the entropy of the calls for those samples, or np.nan if no such calls

Return type List[float]

`trtools.statSTR.`**`GetHWEP`**(*trrecord*, *sample_indexes=[None]*, *uselength=True*)

Compute Hardy Weinberg p-value

Tests whether the number of observed heterozygous vs. homozygous individuals is different than expected under Hardy Weinberg Equilibrium given the observed allele frequencies, based on a binomial test.

Parameters

- **`trrecord`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record that we are computing the statistic for
- **`sample_indexes`** (*`List[Any]`*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **`uselength`** (*`bool`*) – Whether we should collapse alleles by length

Returns **p-value** – The two-sided p-value returned by a binomial test (scipy.stats.binom_test) If there are no calls, return np.nan If the genotype alleles not included in frequencies dictionary, return np.nan One value returned for each sample_index

Return type list of float

`trtools.statSTR.`**`GetHeader`**(*header*, *sample_prefixes*)

Return header items for a column

Parameters

- **`header`** (*`str`*) – Header item
- **`sample_prefixes`** (*`list of str`*) – List of sample prefixes. empty if no sample groups used

Returns **header_items** – List of header items

Return type list of str

`trtools.statSTR.`**`GetHet`**(*trrecord*, *sample_indexes=[None]*, *uselength=True*)

Compute heterozygosity of a locus

Heterozygosity is defined as the probability that two randomly drawn allele are different.

Parameters

- **`trrecord`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record that we are computing the statistic for
- **`sample_indexes`** (*`List[Any]`*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **`uselength`** (*`bool`*) – Whether we should collapse alleles by length

Returns **heterozygosity** – For each sample list, the heterozypostiy of the calls for those samples, or np.nan if no such calls

Return type List[float]

`trtools.statSTR.`**`GetMean`**(*trrecord*, *sample_indexes=[None]*, *uselength=True*)

Compute the mean allele length

Parameters

- **trrecord** (trtools.utils.tr_harmonizer.TRRecord) – The record that we are computing the statistic for
- **sample_indexes** (*List[Any]*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **uselength** (*bool*) –

Returns **mean** – For each sample list, the mean allele length, or np.nan if no calls for that sample

Return type List[float]

trtools.statSTR.**GetMode**(*trrecord*, *sample_indexes=[None]*, *uselength=True*)

Compute the mode of the allele lengths

Parameters

- **trrecord** (*trh.TRRecord object*) – The record that we are computing the statistic for
- **sample_indexes** (*List[Any]*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **uselength** (*bool*) –

Returns **mean** – For each sample list, the mode allele length, or np.nan if no calls for that sample

Return type List[float]

trtools.statSTR.**GetNAlleles**(*trrecord*, *sample_indexes=[None]*, *nalleles_thresh=0.01*, *uselength=True*)

Return allele frequency for a TR

Parameters

- **trrecord** (trtools.utils.tr_harmonizer.TRRecord) – The record that we are computing the statistic for
- **sample_indexes** (*List[Any]*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **nalleles_thresh** (*float*) – The threshold which an allele's frequency must exceed to be counted
- **uselength** (*bool*) – Whether we should collapse alleles by length

Returns Number of called alleles at this locus per sample index. Zero if no alleles were called.

Return type List[int]

trtools.statSTR.**GetNumSamples**(*trrecord*, *sample_indexes=[None]*)

Compute the number of samples

Parameters

- **trrecord** (*trh.TRRecord object*) – The record that we are computing the statistic for
- **sample_indexes** – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.

Returns **numSamples** – The number of samples. One value for each sample list If the allele frequencies dictionary is invalid, return np.nan

Return type list of int

`trtools.statSTR.`**`GetThresh`**(*trrecord*, *sample_indexes=[None]*)

Return the maximum TR allele length observed

Parameters

- **`trrecord`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record that we are computing the statistic for
- **`sample_indexes`** (*`List[Any]`*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.

Returns **thresh** – List of Maximum allele length observed in each sample group, or nan if no alleles called

Return type List[float]

`trtools.statSTR.`**`GetVariance`**(*trrecord*, *sample_indexes=[None]*, *uselength=True*)

Compute the variance of the allele lengths

Parameters

- **`trrecord`** (`trtools.utils.tr_harmonizer.TRRecord`) – The record that we are computing the statistic for
- **`sample_indexes`** (*`List[Any]`*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **`uselength`** (*`bool`*) –

Returns **variance** – For each sample list, the variance of the allele lengths, or np.nan if no calls for that sample

Return type List[float]

`trtools.statSTR.`**`PlotAlleleFreqs`**(*trrecord*, *outprefix*, *sample_indexes=[None]*, *sampleprefixes=None*)

Plot allele frequencies for a locus

Parameters

- **`trrecord`** (*`trh.TRRecord object`*) – The record that we are computing the statistic for
- **`outprefix`** (*`str`*) – Prefix for output file
- **`sample_indexes`** (*`List[Any]`*) – A list of indexes into the numpy rows array to extract subsets of genotypes to stratify over. (e.g. [[True, False, False], [False, True, True]] or [[0], [1,2]] to split three samples into two strata - the first sample and the last two) Can contain None for all samples.
- **`sampleprefixes`** (*`list of str, optional`*) – Prefixes for each sample list to use in legend

`trtools.statSTR.`**`format_nan_precision`**(*precision_format*, *val*)

`trtools.statSTR.`**`getargs`**()

`trtools.statSTR.`**`main`**(*args*)

`trtools.statSTR.`**`run`**`()`

trtools.utils.common module

Common util functions

`trtools.utils.common.`**`MSG`**`(`*msg*, *debug=False*`)`

Write a status message to standard error

Parameters

- **msg** (*str*) – A descriptive message
- **debug** (*bool, optional*) – Only print the message if debug is True

Examples

```
>>> MSG("Something unexpected happened")
```

`trtools.utils.common.`**`WARNING`**`(`*msg*`)`

Write a warning message to standard error

Parameters **msg** (*str*) – A descriptive warning message

Examples

```
>>> WARNING("Something unexpected happened")
```

trtools.utils.mergeutils module

Utilities for reading multiple VCFs simulataneously and keeping them in sync.

`trtools.utils.mergeutils.`**`CheckMin`**`(`*is_min*`)`

Check if we're progressing through VCFs

Parameters **is_min** (*list of bool*) – List indicating if each record is first in sort order

Returns **check** – Set to True if something went wrong

Return type bool

`trtools.utils.mergeutils.`**`CheckPos`**`(`*record*, *chrom*, *pos*`)`

Check a record is at the specified position

Parameters

- **r** (*vcf.Record*) – VCF Record being checked
- **chrom** (*str*) – Chromosome name
- **pos** (*int*) – Chromosome position
- **record** (*cyvcf2.cyvcf2.Variant*) –

Returns **check** – Return True if the current record is at this position

Return type bool

`trtools.utils.mergeutils.`**`DebugPrintRecordLocations`**`(`*current_records*, *is_min*`)`

Debug function to print current records for each file

Parameters

- **current_records** (*list of vcf.Record*) – List of current records from merged files
- **is_min** (*list of bool*) – List of check for if record is first in sort order

Return type None

`trtools.utils.mergeutils.`**`DoneReading`**`(`*records*`)`

Check if all records are at the end of the file

Parameters **records** (*list of vcf.Record*) – List of records from files to merge

Returns **check** – Set to True if all record is None indicating we're done reading the file

Return type list of bool

`trtools.utils.mergeutils.`**`GetAndCheckVCFType`**`(`*vcfs*, *vcftype*`)`

Infer type of multiple VCFs

If they are all the same, return that type If not, return error

Parameters

- **vcfs** (*list of cyvcf2.VCF*) – Multiple VCFs
- **vcftype** (*str*) – If it is unclear which of a few VCF callers produced the underlying VCFs (because the output markings of those VCF callers are similar) this string can be supplied by the user to choose from among those callers.

Returns **vcftype** – Inferred VCF type

Return type str

Raises **TypeError** – If one of the VCFs does not look like it was produced by any supported TR caller, or if one of the VCFs looks like it could have been produced by more than one supported TR caller and vcftype == 'auto', or if, for one of the VCFs, vcftype doesn't match any of the callers that could have produced that VCF, or if the types of the VCFs don't match

`trtools.utils.mergeutils.`**`GetChromOrder`**`(`*r*, *chroms*`)`

Get the chromosome order of a record

Parameters

- **r** (*vcf.Record*) –
- **chroms** (*list of str*) – Ordered list of chromosomes

Returns **order** – Index of r.CHROM in chroms, int Return np.inf if can't find r.CHROM, float

Return type int or float

`trtools.utils.mergeutils.`**`GetChromOrderEqual`**`(`*chrom_order*, *min_chrom*`)`

Check chrom order

Parameters

- **chrom_order** (*int*) – Chromosome order
- **min_chrom** (*int*) – Current chromosome order

Returns **equal** – Return True if chrom_order==min_chrom and chrom_order != np.inf

Return type bool

trtools.utils.mergeutils.**GetIncrementAndComparability**(*record_list*, *chroms*, *overlap_callback=<function default_callback>*)

Get list that says which records should be skipped in the next iteration (increment), and whether they are all comparable / mergable The value of increment elements is determined by the (harmonized) position of corresponding records

Parameters

- **record_list** (*trh.TRRecord*) – list of current records from each file being merged
- **chroms** (*list of str*) – Ordered list of all chromosomes
- **overlap_callback** (*Callable[[List[Optional[trh.TRRecord]], List[int], int], Union[bool, List[bool]]*) – Function that calculates whether the records are comparable

Returns

- **increment** (*list of bool*) – List or bools, where items are set to True when the record at the index of the item should be skipped during VCF file comparison.
- **comparable** (*bool or list of bool*) – Value, that determines whether current records are comparable / mergable, depending on the callback

Return type Tuple[List[bool], Union[bool, List[bool]]]

trtools.utils.mergeutils.**GetMinRecords**(*record_list*, *chroms*)

Check if each record is next up in sort order

Return a vector of boolean set to true if the record is in lowest sort order of all the records Use order in chroms to determine sort order of chromosomes

Parameters

- **record_list** (*list of CYVCF_RECORD*) – list of current records from each file being merged
- **chroms** (*list of str*) – Ordered list of all chromosomes

Returns **checks** – Set to True for records that are first in sort order

Return type list of bool

trtools.utils.mergeutils.**GetNextRecords**(*readers*, *current_records*, *increment*)

Increment readers of each file

Increment readers[i] if increment[i] set to true Else keep current_records[i]

Parameters

- **readers** (*list of vcf.Reader*) – List of readers for all files being merged
- **current_records** (*list of vcf.Record*) – List of current records for all readers
- **increment** (*list of bool*) – List indicating if each file should be incremented

Returns **new_records** – List of next records for each file

Return type list of vcf.Record

trtools.utils.mergeutils.**GetPos**(*r*)

Get the position of a record

Parameters **r** (*vcf.Record*) –

Returns **pos** – If r is None, returns np.inf, which is a float

Return type int

`trtools.utils.mergeutils.`**`GetSamples`**`(`*readers*, *filenames=None*`)`

Get list of samples used in all files being merged

Parameters

- **readers** (*list of cyvcf2.VCF*) –
- **usefilenames** (*optional list of filenames*) – If present, add filename to sample names. Useful if sample names overlap across files Must be the same length as readers
- **filenames** (*Optional[str]*) –

Returns **samples** – List of samples in merged list

Return type list of str

`trtools.utils.mergeutils.`**`GetSharedSamples`**`(`*readers*`)`

Get list of samples used in all files being merged

Parameters **readers** (*list of cyvcf.VCF objects*) –

Returns **samples** – Samples present in all readers

Return type list of str

`trtools.utils.mergeutils.`**`InitReaders`**`(`*readers*`)`

Increment readers of each file

Returns list of first records from list of readers.

Parameters **readers** (*list of cyvcf2.VCF*) – List of readers for all files being merged

Returns List of next records for each file

Return type list of vcf.Record

`trtools.utils.mergeutils.`**`LoadReaders`**`(`*vcffiles*, *region=None*`)`

Return list of VCF readers

Parameters

- **vcffiles** (*list of str*) – List of VCF files to merge
- **region** (*str, optional*) – Chrom:start-end to restrict to

Returns **readers** – VCF readers list for all files to merge

Return type list of vcf.Reader

`trtools.utils.mergeutils.`**`default_callback`**`(`*records*, *chrom_order*, *min_chrom_index*`)`

Parameters

- **records** (*List[trtools.utils.tr_harmonizer.TRRecord]*) –
- **chrom_order** (*List[int]*) –
- **min_chrom_index** (*int*) –

Return type bool

trtools.utils.tr_harmonizer module

Utilities for harmonizing tandem repeat VCF records.

Handles VCFs generated by various TR genotyping tools

`trtools.utils.tr_harmonizer.`**`HarmonizeRecord`**`(`*vcftype*, *vcfrecord*`)`

Create a standardized TRRecord object out of a cyvcf2.Variant object of possibly unknown type.

Parameters

- **vcfrecord** (*cyvcf2.cyvcf2.Variant*) – A cyvcf2.Variant Object
- **vcftype** (*Union[str,* `trtools.utils.tr_harmonizer.VcfTypes`*]*) –

Returns A TRRecord object built out of the input record

Return type *TRRecord*

`trtools.utils.tr_harmonizer.`**`HasLengthAltGenotypes`**`(`*vcftype*`)`

Determine if the alt alleles of variants are given by length.

If True, then alt alleles for all variants produced by this caller are specified by length and not by sequence. Sequences are fabricated according to *`trtools.utils.utils.FabricateAllele()`*.

Returns Indicates whether alt alleles are specified by length

Return type bool

Parameters **vcftype** (*Union[str,* `trtools.utils.tr_harmonizer.VcfTypes`*]*) –

`trtools.utils.tr_harmonizer.`**`HasLengthRefGenotype`**`(`*vcftype*`)`

Determine if the reference alleles of variants are given by length.

If True, then reference alleles for all variants produced by this caller are specified by length and not by sequence. Sequences are fabricated according to *`trtools.utils.utils.FabricateAllele()`*.

If True, then *`HasLengthAltGenotypes()`* will also be true

Returns Indicates whether ref alleles are specified by length

Return type bool

Parameters **vcftype** (*Union[str,* `trtools.utils.tr_harmonizer.VcfTypes`*]*) –

`trtools.utils.tr_harmonizer.`**`InferVCFType`**`(`*vcffile*, *vcftype='auto'*`)`

Infer the genotyping tool used to create the VCF.

When we can, infer from header metadata. Otherwise, try to infer the type from the ALT field.

Parameters

- **vcffile** (*cyvcf2.cyvcf2.VCF*) – The input VCF file
- **vcftype** (*Union[str,* `trtools.utils.tr_harmonizer.VcfTypes`*]*) – If it is unclear which of a few VCF callers produced the underlying VCF (because the output markings of those VCF callers are similar) this string can be supplied by the user to choose from among those callers.

Returns **vcftype** – Type of the VCF file

Return type *VcfTypes*

Raises **TypeError** – If this VCF does not look like it was produced by any supported TR caller, or if it looks like it could have been produced by more than one supported TR caller and vcftype == 'auto', or if vcftype doesn't match any of the callers that could have produced this VCF.

`trtools.utils.tr_harmonizer.`**`IsBeagleVCF`**`(`*vcffile*`)`

Is this a VCF produced by running the Beagle software to impute STRs from a panel generated by an TR genotyper, or does it consist of STRs directly called by a TR genotyper?

Parameters **`vcffile`** (*`cyvcf2.cyvcf2.VCF`*) – The input VCF file

Returns Whether this is a VCF produced by Beagle

Return type bool

`trtools.utils.tr_harmonizer.`**`MayHaveImpureRepeats`**`(`*vcftype*`)`

Determine if any of the alleles in this VCF may contain impure repeats.

Specifically, impure repeats include:

- impurities in the underlying sequence (e.g. AAATAAAAA)
- partial repeats (e.g. AATAATAATAA)

This is a guarantee that the caller attempted to call impure repeats, not that it found any. It also does not guarantee that all impurities present were identified and called.

Returns Indicates whether repeat sequences may be impure

Return type bool

Parameters **`vcftype`** (*`Union[str, trtools.utils.tr_harmonizer.VcfTypes]`*) –

`class trtools.utils.tr_harmonizer.`**`TRRecord`**`(`*vcfrecord*, *ref_allele*, *alt_alleles*, *motif*, *record_id*, *quality_field*, ***, *harmonized_pos=None*, *full_alleles=None*, *ref_allele_length=None*, *alt_allele_lengths=None*, *quality_score_transform=None*`)`

Bases: `object`

A representation of a VCF record specialized for TR fields.

Allows downstream functions to be agnostic to the genotyping tool used to create the record.

Parameters

- **`vcfrecord`** (*`cyvcf2.cyvcf2.Variant`*) – Cyvcf2 Variant object with the underlying data
- **`ref_allele`** (*`Optional[str]`*) – Reference allele string
- **`alt_alleles`** (*`Optional[List[str]]`*) – List of alternate allele strings
- **`motif`** (*`str`*) – Repeat unit
- **`record_id`** (*`str`*) – Identifier for the record
- **`quality_field`** (*`Optional[str]`*) – the name of the FORMAT field which contains the quality score for each call for this record
- **`harmonized_pos`** (*`Optional[int]`*) –
- **`full_alleles`** (*`Optional[Tuple[str, List[str]]]`*) –
- **`ref_allele_length`** (*`Optional[float]`*) –
- **`alt_allele_lengths`** (*`Optional[List[float]]`*) –
- **`quality_score_transform`** (*`Optional[Callable[[...], float]]`*) –

`vcfrecord`

The cyvcf2 Variant object used to init this record.

Type cyvcf2.Variant

ref_allele
Reference allele sequences, fabricated if necessary. Gets converted to uppercase e.g. ACGACGACG

Type str

alt_alleles
List of alternate allele sequences, fabricated if necessary

Type List[str]

motif
Repeat unit

Type str

record_id
Identifier for the record

Type str

chrom
The chromosome this locus is in

Type str

pos
The bp along the chromosome that this locus is at (ignoring flanking base pairs/full alleles)

Type int

end_pos
Position of the last bp of ref allele (ignoring flanking base pairs/full alleles)

full_alleles_pos
Position of the first bp of the full ref allele (including the flanking base pairs)

full_alleles_end_pos
Position of the last bp of the full ref allele (including the flanking base pairs)

info
The dictionary of INFO fields at this locus

Type Dict[str, Any]

format
The dictionary of FORMAT fields at this locus. Numeric format fields are 2D numpy arrays with rows corresponding to samples (normally 1 column, but if there are multiple numbers then more than one column) String format fields are 1D numpy arrays with entries corresponding to samples

Type Dict[str, np.ndarray]

Parameters

- **harmonized_pos** (*Optional[int]*) – If this record has flanking base pairs before the repeat, set this to note at which bp the repeat begins
- **full_alleles** (*Optional[Tuple[str, List[str]]]*) – A tuple of string genotypes (ref_allele, [alt_alleles]) where each allele may contain any number of flanking basepairs in addition to containing the tandem repeat. If set, these can be accessed through *GetFullStringGenotypes()* If the alt alleles have differently sized flanks than the ref allele then those alt alleles will be improperly trimmed.

- **alt_allele_lengths** (*Optional[List[float]]*) – The lengths of each of the alt alleles, in order. Thus is measured in number of copies of repeat unit, NOT the allele length in base pairs.

  Should be passed to the constructor when only the lengths of the alt alleles were measured and not the sequences. If sequences are passed to the constructor then this is set automatically.

  If this is passed, the alt_alleles parameter to the constructor must be set to None and the alt_alleles attribute of the record will be set to fabricated alleles (see *trtools.utils.utils.FabricateAllele()*)
- **ref_allele_length** (*Optional[float]*) – like alt_allele_lengths, but for the reference allele. If this is passed, alt_allele_lengths must also be passed
- **quality_score_transform** (*Optional[Callable[[...], float]]*) – A function which turns the quality_field value into a float score. When None, the quality_field values are assumed to already be floats
- **vcfrecord** (*cyvcf2.cyvcf2.Variant*) –
- **ref_allele** (*Optional[str]*) –
- **alt_alleles** (*Optional[List[str]]*) –
- **motif** (*str*) –
- **record_id** (*str*) –
- **quality_field** (*Optional[str]*) –

Notes

Alleles are stored as upper case strings with all the repeats written out. Alleles may contain partial repeat copies or impurities. This class will attempt to make sure alleles do not contain any extra base pairs to either side of the repeat. If you wish to have those base pairs, use the 'Full' methods

GetAlleleCounts(*sample_index=None*, ***, *uselength=True*, *index=False*, *fullgenotypes=False*)

Get the counts of each allele for a record.

This does not return counts of no calls as it is not clear how many 'no call alleles' would be present per no call

Alleles that are not called in any sample are not present in the returned dictionary

Parameters

- **sample_index** (*Optional[Any]*) – Used to index the numpy array of samples. So can be a numpy array of sample indicies, or a bool array with length of the number of samples, etc. If None, then all samples are included.
- **uselength** (*bool, optional*) – If True, represent alleles a lengths else represent as strings
- **index** (*bool*) – If True, represent alleles as indexes (0 = ref, 1 = first_alt, etc.) instead of sequences or lengths
- **fullgenotypes** (*bool*) – If True, include flanking basepairs in allele representations Only makes sense when expliictly stating uselength=False. Cannot be combined with index.

Returns **allele_counts** – Gives the count of each allele. The type of allele representation is determined by the uselength, index and fullgenotypes optional parameters.

Return type Dict[Any, int]

`GetAlleleFreqs`(*sample_index=None*, *, *uselength=True*, *index=False*, *fullgenotypes=False*)

Get the frequencies of each allele for a record.

Parameters

- **sample_index** (*Optional[Any]*) – Used to index the numpy array of samples. So can be a numpy array of sample indicies, or a bool array with length of the number of samples, etc. If None, then all samples are included.
- **uselength** (*bool*) – If True, represent alleles a lengths else represent as strings
- **index** (*bool*) – If True, represent alleles as indexes (0 = ref, 1 = first_alt, etc.) instead of sequences or lengths
- **fullgenotypes** (*bool*) – If True, include flanking basepairs in allele representations. Only makes sense when expliictly stating uselength=False. Cannot be combined with index.

Returns **allele_freqs** – Gives the frequency of each allele among called samples The type of allele representation is determined by the uselength, index and fullgenotypes optional parameters.

Return type Dict[Any, float]

`GetCallRate`(*strict=True*)

Return the call rate at this locus.

Parameters **strict** (*bool*) – By default genotypes such as '1/.' are considered not called because at least one of the haplotypes present is not called. Set strict = False to mark these as being called. Note: genotypes having lesser ploidy will not be marked as no calls even when strict = True (e.g. if some samples have tetraploid genotypes at this locus, a genotype of '1/2/2' will be marked as called even though it is triploid)

Returns

- *The fraction of the samples at this locus that have been*
- *called. If there are no samples in the vcf this record comes from*
- *then return np.nan instead*

Return type float

`GetCalledSamples`(*strict=True*)

Get an array listing which samples have been called at this locus.

Parameters **strict** (*bool*) – By default genotypes such as '1/.' are considered not called because at least one of the haplotypes present is not called. Set strict = False to mark these as being called. Note: genotypes having lesser ploidy will not be marked as no calls even when strict = True (e.g. if some samples have tetraploid genotypes at this locus, a genotype of '1/2/2' will be marked as called even though it is triploid)

Returns A bool array of length equal to the number of samples, where true indicates a sample has been called and false indicates otherwise. If there are no samples in the vcf this record comes from then return None instead

Return type Optional[np.ndarray]

`GetFullStringGenotypes`()

Get an array of full string genotypes for each sample. See `GetStringGenotypes()` for details and limitations of string genotypes.

If the sample does not have full genotypes that are distinct from its regular string genotypes (because no flanking base pairs were called) then the regular string genotypes are returned.

Returns The numpy array described above, of type '<UN' where 'N' is the max allele length. If there are no samples in the vcf this record comes from then return None instead

Return type Optional[np.ndarray]

`GetGenotypeCounts`(*sample_index=None*, *uselength=True*, *index=False*, *fullgenotypes=False*, *include_nocalls=False*)

Get the counts of each genotype for a record.

For samples with a lower ploidy than the max ploidy among all samples, the -2 placeholder haplotypes are sorted to the beginning of the call (e.g. (-2, 5) instead of (5, -2))

This currently returns unphased genotypes (with no phasing column), it could be extend to have an option to respect phasing

Parameters

- **sample_index** (*Optional[Any]*) – Used to index the numpy array of samples. So can be a numpy array of sample indicies, or a bool array with length of the number of samples, etc. If None, then all samples are included.
- **uselength** (*bool*) – If True, represent alleles as lengths else represent as strings
- **index** (*bool*) – If True, represent alleles as indexes (0 = ref, 1 = first_alt, etc.) instead of sequences or lengths
- **fullgenotypes** (*bool*) – If True, include flanking basepairs in allele representations. Only makes sense when expliictly stating uselength=False. Cannot be combined with index.
- **include_nocalls** (*bool*) – If False, all genotypes with one or more uncalled haplotypes (-1 or '.') are excluded from the returned dictionary, they are included if True. Genotypes with lower ploidy (-2 or ',') are included regardless.

Returns **genotype_counts** – Gives the count of each genotype. Genotypes are represented as tuples of alleles, where the type of allele representation is determined by the uselength, index and fullgenotypes optional parameters.

Return type Dict[tuple, int]

`GetGenotypeIndicies`()

Get an array of genotype indicies across all samples.

A genotype index is a number 0, 1, 2 … where 0 corresponds to the reference allele, 1 to the first alt allele, 2 to the second, etc. The array is an array of ints with one row per sample. The number of columns is the maximum ploidy of any sample (normally 2) plus 1 for phasing. All but the final column represent the index of the genotypes of each call. The final column has values 0 for unphased sampels or 1 for phased. So a sample with gt '0|2' would be represented by the row [0, 2, 1] and a sample with gt '3/0' would be represented by the row [3, 0, 0]. Uncalled haplotypes (represented by '.' in the VCF) are represented by '-1' genotypes. If the sample has fewer haplotypes than the maximum ploidy of all samples at this locus, then the row is padded with -2s, so a haploid sample with gt '1' where other samples are diploid would be represented by the row [1, -2, 0]. If all the genotype columns for a sample are negative then the sample is a no call. Note: the value of the phasing column is unspecified for haploid or no-call samples.

Returns The numpy array described above, of type int. If there are no samples in the vcf this record comes from then return None instead

Return type Optional[np.ndarray]

`GetLengthGenotypes`()

Get an array of length genotypes for each sample.

Represents the sample's genotype in terms of the number of repeats of the motif in each allele. Returns a pair of floats - alleles including partial repeats or other impurities may have noninteger lengths.

The array is as described in `GetGenotypeIndicies()` except that indicies are replaced by their length genotypes. -1s, -2s and the phasing bits are not modified.

For records with both regular and full sequences (those with flanking bps), this returns the length of the regular sequences

Returns The numpy array described above, of type float If there are no samples in the vcf this record comes from then return None instead

Return type Optional[np.ndarray]

`GetMaxAllele`(*sample_index=None*)

Get the maximum allele length called in a record.

Represents lengths in terms of the number of repeats of the motif. The longest allele may have a noninteger length if it includes partial repeats or other impurities.

For records with both regular and full sequences (those with flanking bps), this returns the length of the regular sequences

Parameters **sample_index** (*Optional[Any]*) – Used to index the numpy array of samples. So can be a numpy array of sample indicies, or a bool array with length of the number of samples, etc. If None, then all samples are included.

Returns **maxallele** – The maximum allele length called (in number of repeat units), or nan if no alleles called

Return type float

`GetMaxPloidy()`

Return the maximum ploidy of any sample at this locus.

All genotypes will be a tuple of that many haplotypes, where samples with a smaller ploidy than that will have haplotypes at the end of the tuple set to ',' (for string genotypes) or -2 (for index or length genotypes)

Return type int

`GetNumSamples()`

Return the number of samples at this locus (called or not).

Same as the number of samples in the overall vcf

Return type int

`GetQualityScores()`

Get the quality scores of the calls for each sample.

The meaning and reliability of these scores is genotyper dependent, see the doc section *Quality Scores*.

Returns An array of quality score floats, one row per sample Samples which were not called have the value np.nan

Return type np.ndarray

`GetSamplePloidies()`

Get an array listing the ploidies of each sample

Returns An array of positive ints with length equal to the number of samples where each entry denotes the number of genotypes for each sample at this locus (including no calls) If there are no samples in the vcf this record comes from then return None instead

Return type Optional[np.ndarray]

GetStringGenotypes()

Get an array of string genotypes for each sample.

The array is as described in *GetGenotypeIndicies()* except that the indicies are replaced by their corresponding sequences, -1 indicies (nocalls) are replaced by '.', -2 indicies (not present due to smaller ploidy) are replaced by ',', and the phasing bits (0 or 1) are replaced by the strings '0' or '1'.

Will not include flanking base pairs. To get genotypes that include flanking base pairs (for callers that call those), use *GetFullStringGenotypes()*. For callers that include flanking base pairs it is possible that some of the alleles in the regular string genotypes (with the flanks stripped) will be identical. In this case, you may use *UniqueStringGenotypeMapping()* to get a canonical unique subset of indicies which represent all possible alleles.

Note that some TR callers will only call allele lengths, not allele sequences. In such a case, this method will return a fabricated sequence based on the called length (see *trtools.utils.utils.FabricateAllele()*) and a warning will be raised. This may not be intended - use *GetLengthGenotypes()* for a fully caller agnostic way of handling genotypes.

This method is inefficient for many samples, consider either using length genotypes (*GetLengthGenotypes()*), or using genotype indicies (*GetGenotypeIndicies()*) and accessing string genotypes as needed through the fields ref_allele and alt_alleles, instead.

Returns The numpy array described above, of type '<UN' where 'N' is the max allele length. If there are no samples in the vcf this record comes from then return None instead

Return type Optional[np.ndarray]

HasFabricatedAltAlleles()

Determine if this record has fabricated alt alleles.

Returns True iff alt_allele_lengths was passed to this record's constructor.

Return type bool

HasFabricatedRefAllele()

Determine if this record has a fabricated ref allels.

Returns True iff ref_allele_length was passed to this record's constructor.

Return type bool

HasFullStringGenotypes()

Determine if this record has full string genotypes.

Returns True iff *GetFullStringGenotypes()* will return a different value than *GetStringGenotypes()* for some alleles.

Return type bool

HasQualityScores()

Does this TRRecord contain quality scores for each of its calls? If present, the meaning and reliability of these scores is genotyper dependent, see the doc section *Quality Scores*.

Returns Whether or not a FORMAT field that could be interpreted as a quality score has been identified

Return type boolean

UniqueLengthGenotypeMapping()

Get a mapping whose values are unique string genotype indicies.

Returns **genotypeMapping** – A mapping allele idx -> allele idx whose keys are all allele indicies and whose values are a subset of indicies which represents all the unique length alleles for this variant. For variants where multiple alleles have the same length, all will map to a single index from among those alleles.

Return type Dict[int, int]

`UniqueLengthGenotypes()`

Find allele indicies corresponding to the unique length alleles.

Equivalent to calling `set(UniqueLengthGenotypeMapping().values())`

Returns The indicies of the unique string alleles

Return type Set[int]

`UniqueStringGenotypeMapping()`

Get a mapping whose values are unique string genotype indicies.

Returns A mapping allele idx -> allele idx whose keys are all allele indicies and whose values are a subset of indicies which represents all the unique regular string alleles for this variant. For almost all records, this will be a mapping from each index to itself. For some records with full string genotypes that include flanking base pairs, some of the regular string alleles will be identical. In this case, only one of those allele's indicies will be in the set of values of this dictionary, and all identical alleles will map to that one index.

Return type Dict[int, int]

`UniqueStringGenotypes()`

Find allele indicies corresponding to the unique alleles.

Equivalent to calling `set(UniqueStringGenotypeMapping().values())`

Returns The indicies of the unique string alleles

Return type Set[int]

`_CheckRecord()`

Check that this record is properly constructed.

Checks that the same number of alt alleles were specified as in the underlying record and that the full_alleles, if supplied, contain their corresponding standard alleles

Raises an error if a check fails

`class trtools.utils.tr_harmonizer.TRRecordHarmonizer(`*vcffile*, *vcftype='auto'*`)`

Bases: `object`

Class producing a uniform interface for accessing TR VCF records.

Produces the same output interface regardless of the tool that created the input VCF.

The main purpose of this class is to infer which tool a VCF came from, and appropriately convert its records to TRRecord objects.

This class provides the object oriented paradigm for iterating through a TR vcf. If you wish to use the functional paradigm and provide the cyvcf2.Variant objects yourself, use the top-level functions in this module.

Parameters

- **vcffile** (*cyvcf2.VCF instance*) –
- **vcftype** (*{'auto', 'gangstr', 'advntr', 'hipstr', 'eh', 'popstr'}, optional*) – Type of the VCF file. Default='auto'. If vcftype=='auto', attempts to infer the type.

`vcffile`

Type cyvcf2.VCF instance

vcftype

Type of the VCF file. Must be included in VcfTypes

Type enum

Raises **TypeError** – If the type of the VCF cannot be properly inferred. See *InferVCFType()* for more details.

Parameters

- **vcffile** (*cyvcf2.cyvcf2.VCF*) –
- **vcftype** (*Union[str,* trtools.utils.tr_harmonizer.VcfTypes*]*) –

HasLengthAltGenotypes()

Determine if the alt alleles of variants are given by length.

See also:

`tr_harmonizer.HasLengthAltGenotypes`

Return type bool

HasLengthRefGenotype()

Determine if the reference alleles of variants are given by length.

See also:

`tr_harmonizer.HasLengthRefGenotype`

Return type bool

HasQualityScore()

Does this VCF contain quality scores for each of its calls? If present, the meaning and reliability of these scores is genotyper dependent, see the doc section *Quality Scores*.

Returns Whether or not a FORMAT field that could be interpreted as a quality score has been identified

Return type bool

IsBeagleVCF()

Is this a VCF produced by running the Beagle software to impute STRs from a panel generated by an TR genotyper?

See also:

`tr_harmonizer.IsBeagleVCF`

Return type bool

MayHaveImpureRepeats()

Determine if any of the alleles in this VCF may contain impure repeats.

See also:

`tr_harmonizer.MayHaveImpureRepeats`

Return type bool

class trtools.utils.tr_harmonizer.**VcfTypes**(*value*)

Bases: `enum.Enum`

The different tr callers that tr_harmonizer supports.

advntr = 'advntr'

eh = 'eh'

gangstr = 'gangstr'

hipstr = 'hipstr'

popstr = 'popstr'

class trtools.utils.tr_harmonizer.**_Cyvcf2FormatDict**(*record*)

Bases: `object`

Provide an immutable dict-like interface for accessing format fields from a cyvcf2 record. To iterate over this dict, use `iter(this)` or `this.keys()`.

Parameters **record** (*cyvcf2.cyvcf2.Variant*) –

get(*key*)

Parameters **key** (*str*) –

keys()

trtools.utils.tr_harmonizer.**_HarmonizeAdVNTRRecord**(*vcfrecord*)

Turn a cyvcf2.Variant with adVNTR content into a TRRecord.

Parameters **vcfrecord** (*cyvcf2.cyvcf2.Variant*) – A cyvcf2.Variant Object

Returns

Return type *TRRecord*

trtools.utils.tr_harmonizer.**_HarmonizeEHRecord**(*vcfrecord*)

Turn a cyvcf2.Variant with EH content into a TRRecord.

Parameters **vcfrecord** (*cyvcf2.cyvcf2.Variant*) – A cyvcf2.Variant Object

Returns

Return type *TRRecord*

trtools.utils.tr_harmonizer.**_HarmonizeGangSTRRecord**(*vcfrecord*)

Turn a cyvcf2.Variant with GangSTR content into a TRRecord.

Parameters **vcfrecord** (*cyvcf2.cyvcf2.Variant*) – A cyvcf2.Variant Object

Returns

Return type *TRRecord*

trtools.utils.tr_harmonizer.**_HarmonizeHipSTRRecord**(*vcfrecord*)

Turn a cyvcf2.Variant with HipSTR content into a TRRecord.

Parameters **vcfrecord** (*cyvcf2.cyvcf2.Variant*) – A cyvcf2.Variant Object

Returns

Return type *TRRecord*

trtools.utils.tr_harmonizer.**_HarmonizePopSTRRecord**(*vcfrecord*)

Turn a cyvcf2.Variant with popSTR content into a TRRecord.

Parameters **`vcfrecord`** (*`cyvcf2.cyvcf2.Variant`*) – A cyvcf2.Variant Object

Returns

Return type *TRRecord*

trtools.utils.utils

Util functions for calculating summary STR statistics and performing basic string operations on STR alleles.

class `trtools.utils.utils.`**`ArgumentDefaultsHelpFormatter`**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: `argparse.HelpFormatter`

Build a custom argument parser that works just like argparse.ArgumentDefaultsHelpFormatter except that it doesn't display None defaults. Everything below is copied from the python library source except for that.

Help message formatter which adds default values to argument help.

Only the name of this class is considered a public API. All the methods provided by the class are considered an implementation detail.

`trtools.utils.utils.`**`FabricateAllele`**(*motif*, *length*)

Fabricate an allele with the given motif and length.

Parameters

- **`motif`** (*`str`*) – the motif to build the allele from
- **`length`** (*`float`*) – the number of times to copy the motif (noninteger implies partial repeats). This does NOT specify the desired length of the returned string.

Returns the fabricated allele string

Return type str

Notes

The allele is fabricated with the given motif orientation (e.g. motif = 'ACG' could produce 'ACGACGACG' but not 'CGACGACGA'). Fabricated alleles will contain no impurities nor flanking base pairs. In the case of length being a noninteger float (because of partial repeats) and where it is unclear if the last nucleotide should be included in the fabricated repeat or not due to imprecision in the length float, the last nucleotide will always be left off (the length of the returned string will always be rounded down).

`trtools.utils.utils.`**`GetCanonicalMotif`**(*repseq*)

Get canonical STR sequence, considering both strands

The canonical sequence is the first alphabetically out of all possible rotations on + and - strands of the sequence. e.g. "TG" canonical sequence is "AC".

Parameters **`repseq`** (*`str`*) – String giving a STR motif (repeat unit sequence)

Returns **canon** – The canonical sequence of the STR motif

Return type str

Examples

```
>>> GetCanonicalMotif("TG")
'AC'
```

`trtools.utils.utils.`**`GetCanonicalOneStrand`**`(`*repseq*`)`

Get canonical STR sequence, considering one strand

The canonical sequence is the first alphabetically out of all possible rotations. e.g. CAG -> AGC.

Parameters **`repseq`** (*`str`*) – String giving a STR motif (repeat unit sequence)

Returns **canon** – The canonical sequence of the STR motif

Return type str

Examples

```
>>> GetCanonicalOneStrand("CAG")
'AGC'
```

`trtools.utils.utils.`**`GetContigs`**`(`*vcf*`)`

Returns the contig IDs in the VCF.

Parameters **`vcf`** (*`cyvcf2.cyvcf2.VCF`*) – The vcf to get contigs from

Returns A list of contig IDs

Return type List[str]

`trtools.utils.utils.`**`GetEntropy`**`(`*allele_freqs*`)`

Compute the (bit) entropy of a locus

Entropy is defined as the *entropy <https://en.wikipedia.org/wiki/Information_content>*_ of the distribution of allele frequencies.

Parameters **`allele_freqs`** (*`dict of object:  float`*) – Dictionary of allele frequencies for each allele. Alleles are typically strings (sequences) or floats (num. repeats)

Returns **entropy** – The entropy of the locus. If the allele frequencies dictionary is invalid, return np.nan

Return type float

Notes

Entropy is computed as:

$$E = -\sum_{i=1..n} -p_i * log_2(p_i)$$

where *p_i* is the frequency of allele *i* and *n* is the number of alleles.

Examples

```
>>> GetEntropy({0:0.5, 1:0.5})
1.0
```

`trtools.utils.utils.`**`GetHardyWeinbergBinomialTest`**`(`*allele_freqs*, *genotype_counts*`)`

Compute Hardy Weinberg p-value

Tests whether the number of observed heterozygous vs. homozygous individuals is different than expected under Hardy Weinberg Equilibrium given the observed allele frequencies, based on a binomial test.

Parameters

- **`allele_freqs`** (*`dict of object: float`*) – Dictionary of allele frequencies for each allele. Alleles are typically strings (sequences) or floats (num. repeats)
- **`genotype_counts`** (*`dict of (object, object): int`*) – Dictionary of counts of each genotype. Genotypes are defined as tuples of alleles. Alleles must be the same as those given in allele_freqs

Returns **p-value** – The two-sided p-value returned by a binomial test (scipy.stats.binom_test) If the allele frequencies dictionary is invalid, return np.nan If genotype alleles not included in frequencies dictionary, return np.nan

Return type float

`trtools.utils.utils.`**`GetHeterozygosity`**`(`*allele_freqs*`)`

Compute heterozygosity of a locus

Heterozygosity is defined as the probability that two randomly drawn alleles are different.

Parameters **`allele_freqs`** (*`dict of object: float`*) – Dictionary of allele frequencies for each allele. Alleles are typically strings (sequences) or floats (num. repeats)

Returns **heterozygosity** – The heterozygosity of the locus. If the allele frequencies dictionary is invalid, return np.nan

Return type float

Notes

Heterzygosity is computed as:

$$H = 1 - \sum_{i=1..n} p_i^2$$

where *p_i* is the frequency of allele *i* and *n* is the number of alleles.

Examples

```
>>> GetHeterozygosity({0:0.5, 1:0.5})
0.5
```

`trtools.utils.utils.`**`GetHomopolymerRun`**`(`*seq*`)`

Compute the maximum homopolymer run length in a sequence

Parameters **`seq`** (*`str`*) – String giving a sequence of nucleotides

Returns **runlength** – The length of the longest homopolymer run

Return type int

Examples

```
>>> GetHomopolymerRun("AATAAAATAAAAAT")
5
```

`trtools.utils.utils.`**`GetMean`**`(`*allele_freqs*`)`

Compute the mean allele length

Parameters **`allele_freqs`** (*`dict of object: float`*) – Dictionary of allele frequencies for each allele. Alleles must be given in lengths (numbers, not strings)

Returns **mean** – Return mean if allele frequencies dictionary is valid

Return type float

Examples

```
>>> GetMean({0:0.5, 1:0.5})
0.5
```

`trtools.utils.utils.`**`GetMode`**`(`*allele_freqs*`)`

Compute the mode allele length.

If more than one allele has the maximum number of copies out of all alleles, choose one at random (but reproducibly)

Parameters **`allele_freqs`** (*`dict of object: float`*) – Dictionary of allele frequencies for each allele. Alleles must be given in lengths (numbers, not strings)

Returns **mode** – Return mode if allele frequencies dictionary is valid

Return type float

Examples

```
>>> GetMode({0:0.1, 1:0.9})
1
```

`trtools.utils.utils.`**`GetVariance`**`(`*allele_freqs*`)`

Compute the variance of the allele lengths

Parameters **`allele_freqs`** (*`dict of object: float`*) – Dictionary of allele frequencies for each allele. Alleles must be given in lengths (numbers, not strings)

Returns **variance** – Return variance if allele frequencies dictionary is valid np.nan otherwise.

Return type float

Examples

```
>>> GetVariance({0:1})
0
```

`trtools.utils.utils.`**`InferRepeatSequence`**`(`*seq*, *period*`)`

TODO change to dynamic programming approach Infer the repeated sequence in a string

Parameters

- **seq** (*str*) – A string of nucleotides
- **period** (*int*) – Length of the repeat unit

Returns **repseq** – The inferred repeat unit (motif). If the input sequence is smaller than the period, repseq consists of all N's.

Return type str

Examples

```
>>> InferRepeatSequence('ATATATAT', 2)
'AT'
```

`trtools.utils.utils.`**`LoadReaders`**`(`*vcf_locs*, *checkgz=True*`)`

Return a list of VCF readers

Parameters

- **vcf_locs** (*List[str]*) – A list of vcf locations
- **checkgz** (*bool*) – Check if each VCF file is gzipped and indexed

Returns **readers** – A list of VCF readers, or None if any of them could not be found. If checkgz, then also return None if any of the VCF readers were not gzipped and tabix indexed.

Return type Optional[List[cvycf2.VCF]]

`trtools.utils.utils.`**`LoadSingleReader`**`(`*vcf_loc*, *checkgz=True*`)`

Return a VCF reader

Parameters

- **vcf_loc** (*str*) – The location of the VCF file to read
- **checkgz** (*bool*) – Check whether VCF file is gzipped and indexed, if not return None

Returns **reader** – The cyvcf2.VCF instance, or None if the VCF is not present or could not be opened

Return type Optional[cyvcf2.VCF]

`trtools.utils.utils.`**`LongestPerfectRepeat`**`(`*seq*, *motif*, *check_reverse=True*`)`

Determine the length (bp) of the longest perfect repeat stretch

Credit: function originally written by Helyaneh Ziaei-Jam

Parameters

- **seq** (*str*) – Repeat region sequence
- **motif** (*str*) – Repeat unit sequence
- **check_reverse** (*bool (optional)*) – If False, don't check reverse complement

Returns **max_match** – Number of bp of longest perfect stretch

Return type int

`trtools.utils.utils.`**`ReverseComplement`**`(`*seq*`)`

Get reverse complement of a sequence.

Converts everything to uppsercase.

Parameters **`seq`** (*`str`*) – String of nucleotides.

Returns **revcompseq** – Reverse complement of the input sequence.

Return type str

Examples

```
>>> ReverseComplement("AGGCT")
'AGCCT'
```

`trtools.utils.utils.`**`ValidateAlleleFreqs`**`(`*allele_freqs*`)`

Check that the allele frequency distribution is valid.

Allele frequencies must sum to 1.

Parameters **`allele_freqs`** (*`dict of object:  float`*) – Dictionary of allele frequencies for each allele. Alleles are typically strings (sequences) or floats (num. repeats)

Returns **is_valid** – Return True if the distribution is valid, else False

Return type bool

Examples

```
>>> ValidateAlleleFreqs({0:0.5, 1:0.5})
True
```

10.5 Site Indices

- General Index
- Python Module Index

PYTHON MODULE INDEX

INDEX