tspop

Release 0.1

Georgia Tsambos

Apr 13, 2023

CONTENTS

`tspop` is available on PyPi for installation with `pip`:

```
pip install tspop
```

If you absolutely must use the most up-to-date code, you can do it by cloning the `git` repository,

```
git clone https://github.com/gtsambos/tspop
```

navigating into the root directory,

```
cd tspop
```

and installing it like this:

```
pip install .
```

CHAPTER ONE

DEVELOPER INSTALLATION

To install `tspop` in addition to the packages needed to develop and run tests, perform the first two steps above, then run the following command:

```
pip install -e .[dev]
```

I recommend developing *tspop* in a virtual environment like a [conda environment](https://conda.io/projects/conda/en/latest/index.html).

CHAPTER

TWO

RUNNING THE TESTS

The test suite uses the [*pytest*](https://docs.pytest.org/en/7.2.x/) module.

```
pytest tests
```

You can run specific classes or tests in specific test files:

```
pytest tests/test_tspop.py::TestIbdSquash
```

To get printed output from the tests, use the *s* flag:

```
pytest -s tests/test_tspop.py::TestIbdSquash.test_basic
```

```
pytest tests/test_tspop.py::TestIbdSquash.test_basic
```

CHAPTER THREE

COMPILING THE DOCUMENTATION

Note: Finish later.

```
cd docs
make clean
make html
```

CHAPTER

FOUR

SIMULATION SETUP

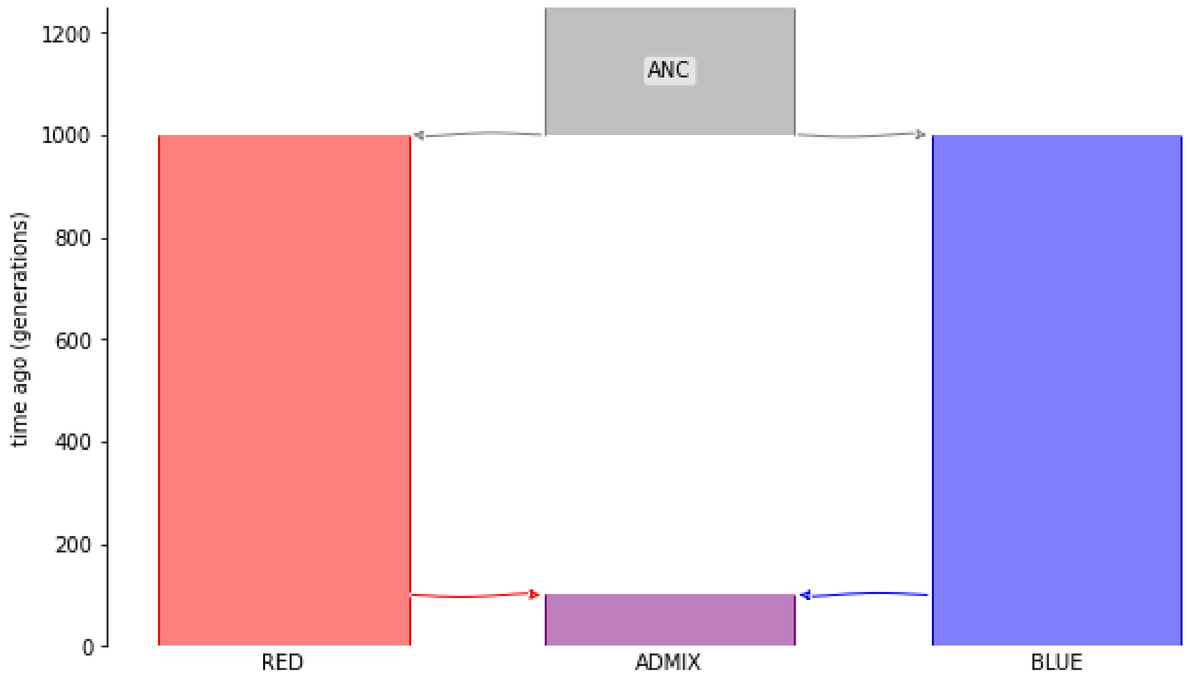
Population-based ancestry is only well-defined with reference to some particular time. For instance, suppose that my maternal grandmother belonged to Population A, and my mother migrated into Population B. I inherited some of my genome from these ancestors – but which population did it come from? The answer depends on a point in *time* – and in particular, on whether we are interested in ancestry one or two generations ago.

By default, `msprime` and `SLiM` do not retain information about the ancestry of individuals at each timepoint in the simulated history. The rest of this page explains how to rectify this.

4.1 `msprime` simulations

Let's make this more concrete with an example.

Suppose there was an admixture event between two populations, *RED* and *BLUE*, that produces an admixed population *ADMIX*. The admixture happened 100 generations ago. These two populations split from a common ancestral population, *ANC*. The split was 1000 generations ago.



Here's a `msprime.Demography` object describing this demographic scenario:

```
import msprime

pop_size = 500
demography = msprime.Demography()
demography.add_population(name="RED", initial_size=pop_size)
demography.add_population(name="BLUE", initial_size=pop_size)
demography.add_population(name="ADMIX", initial_size=pop_size)
demography.add_population(name="ANC", initial_size=pop_size)
demography.add_admixture(
    time=100, derived="ADMIX", ancestral=["RED", "BLUE"], proportions=[0.5, 0.5]
)
demography.add_population_split(
    time=1000, derived=["RED", "BLUE"], ancestral="ANC"
)
```

We'll simulate a 100kb genomic region for two diploid individuals from the admixed population *ADMIX*:

```
ts = msprime.sim_ancestry(
    samples={"RED": 0, "BLUE": 0, "ADMIX" : 2},
    demography=demography,
    random_seed=1011,
    sequence_length=100000,
    recombination_rate=3e-8
)
```

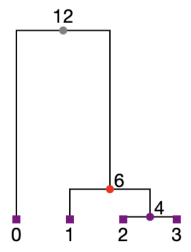
Have a look at this msprime tutorial if you need a refresher on this syntax.

By default, the tree sequences generated by *msprime* will only show the ancestral haplotypes that happen to be coalescent ancestors. On their own, this is not sufficient to get information about population-based ancestry at all locations along the genome.

For instance, here is the first tree in the tree sequence generated by the simulation above. This tree describes the genealogical relationships between the samples on the leftmost part of the simulated genome:

```
# Note that this code will only work in a Jupyter notebook
from IPython.display import SVG

colour_map = {0:"red", 1:"blue", 2: "purple", 3: "gray"}
node_colours = {u.id: colour_map[u.population] for u in ts.nodes()}
tree = ts.first()
SVG(tree.draw(node_colours=node_colours))
```



From this, we see that samples 1, 2 and 3 have ancestry with the red population at this location in their genomes. However, we cannot be sure about the provenance of sample 0 based on the information displayed here. At this genomic location, sample 0 is very deeply diverged from the other samples. In fact, it is so deeply diverged that it's most recent coalescence with the other samples (at node 12) pre-dates the 'split' between the red and blue ancestral populations. To see which population it has inherited from at this location, we'd need to 'mark' one of the more recent (non-coalescent) ancestors of sample 0 to retain.

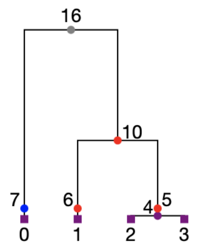
The `msprime.Demography.add_census()` method (documented here) is a special demographic event that we added into `msprime` to do precisely this. More specifically, `add_census` records a node on all lineages that are extant at some user-specified time in the simulation. This is needed to simulate complete information about local ancestry.

The code below is the same that we specified above, but with a census event at `time=100.001`. Note that this time is just before the admixture event creating population `ADMIX`.

```
# Make the Demography object.
demography = msprime.Demography()
demography.add_population(name="RED", initial_size=pop_size)
demography.add_population(name="BLUE", initial_size=pop_size)
demography.add_population(name="ADMIX", initial_size=pop_size)
demography.add_population(name="ANC", initial_size=pop_size)
demography.add_admixture(
    time=100, derived="ADMIX", ancestral=["RED", "BLUE"], proportions=[0.5, 0.5]
)
demography.add_census(time=100.01) # Census is here!
demography.add_population_split(
    time=1000, derived=["RED", "BLUE"], ancestral="ANC"
)

# Simulate.
ts = msprime.sim_ancestry(
    samples={"RED": 0, "BLUE": 0, "ADMIX" : 2},
    demography=demography,
    random_seed=1011,
    sequence_length=100000,
    recombination_rate=3e-8
)
```

Here is a diagram of the first tree in the tree sequence returned by this simulation.



Note that there is now a node on *every* branch in the trees at the time specified in our census event. (In the tree above, these are nodes 5, 6 and 7). This is the information required to extract full information about population-based ancestry at all genomic locations in all samples. For instance, we see here that sample 0 has local ancestry with the blue population, while the other samples have ancestry with the red population.

4.2 SLiM simulations

Use a `treeSeqRememberIndividuals()` call to select census individuals.

```
initialize() {
        initializeTreeSeq();
        initializeMutationRate(0);
        initializeMutationType("m1", 0.5, "f", 0.0);
        initializeGenomicElementType("g1", m1, 1.0);
        initializeGenomicElement(g1, 0, 99999);
        initializeRecombinationRate(3e-8);
}
1 early() {
        sim.addSubpop("p3", 500); // "ANC"
}
1000 early() {
        sim.addSubpop("p0", 500); // "RED"
        sim.addSubpop("p1", 500); // "BLUE"
        p0.setMigrationRates(p3, 1.0);
        p1.setMigrationRates(p3, 1.0);
        p3.setSubpopulationSize(0);
}
1899 late() {
        // The 'census' event:
        // note these individuals have time 101 in the output
        sim.treeSeqRememberIndividuals(sim.subpopulations.individuals);
}
1900 early() {
        sim.addSubpop("p2", 500); // "ADMIX"
        p2.setMigrationRates(c(p0, p1), c(0.5, 0.5));
}
1901 early() {
        // admixture happens in a single generation
        p2.setMigrationRates(c(p0, p1), c(0, 0));
}
2000 late() {
        sim.treeSeqOutput("slim.trees");
}
```

4.3 When should you add the census?

4.3.1 msprime simulations

You should specify the census event at a time when

1. All of the relevant ancestral populations are active.
2. It is unlikely that all samples have coalesced anywhere.
3. There are no other coalescent nodes.

In the example above, condition 1 suggests that we should choose a census time between 100 and 1000 generations in the past. Before this time, the populations did not 'exist', and after this time, the ancestors of the sample were already admixed. To make condition 2 as likely as possible, we should choose a time closer to 100 and 1000. The chosen time of `100.01` satisfies both of these conditions so far. Since we are running a (default) coalescent simulation here, condition 3 is unlikely to be an issue.

Note: Condition 3 is most important when you are running a DTWF simulation. In this situation, you want to avoid placing the census nodes 'on top' of the existing ancestors that are generated at discrete times, so a non-integer time is most suitable here.

4.3.2 `SLiM` simulations

You'll usually want to place the `treeSeqRememberIndividuals()` call in the generation before admixture begins.

CHAPTER

FIVE

BASIC USAGE

Note: **Ensure** that your simulated tree sequence follows the guidelines mentioned in *Simulation setup*.

Here's a sample tree sequence simulated with msprime. Note the census time at 100.01:

```
import msprime

pop_size = 500
sequence_length = 1e7
seed = 98765
rho = 3e-8

# Make the Demography object.
demography = msprime.Demography()
demography.add_population(name="RED", initial_size=pop_size)
demography.add_population(name="BLUE", initial_size=pop_size)
demography.add_population(name="ADMIX", initial_size=pop_size)
demography.add_population(name="ANC", initial_size=pop_size)
demography.add_admixture(
    time=100, derived="ADMIX", ancestral=["RED", "BLUE"], proportions=[0.5, 0.5]
)
demography.add_census(time=100.01) # Census is here!
demography.add_population_split(
    time=1000, derived=["RED", "BLUE"], ancestral="ANC"
)

# Simulate.
ts = msprime.sim_ancestry(
    samples={"RED": 0, "BLUE": 0, "ADMIX" : 2},
    demography=demography,
    random_seed=seed,
    sequence_length=sequence_length,
    recombination_rate=rho
)
```

Apply `tspop.get_pop_ancestry()` to get a `tspop.PopAncestry` object.

```
import tspop

pa = tspop.get_pop_ancestry(ts, census_time=100.01)
```

Use print to see a summary of the information held within the object.

```
print(pa)

> PopAncestry summary
>
> Number of ancestral populations:    2
> Number of sample chromosomes:       4
> Number of ancestors:          118
> Total length of genomes:      40000000.000000
> Ancestral coverage:        40000000.000000
```

The ancestral information itself is inside two tables. The *tspop.PopAncestry.squashed_table* shows tracts of ancestry:

```
st = pa.squashed_table
print(st)

>     sample       left       right  population
> 0        0        0.0    419848.0           0
> 1        0   419848.0    483009.0           1
> 2        0   483009.0   1475765.0           0
> 3        0  1475765.0   2427904.0           1
> 4        0  2427904.0   3635390.0           0
> ..     ...        ...         ...         ...         ...
> 55       3  7369409.0   7596783.0           1
> 56       3  7596783.0   8289015.0           0
> 57       3  8289015.0   8918727.0           1
> 58       3  8918727.0  10000000.0           0
```

The *tspop.PopAncestry.ancestry_table* shows a superset of this information: tracts of ancestry, and the ancestor at the census time who contributed each tract. Each row of the squashed table above can be obtained by 'gluing together' rows of the ancestry table.

```
at = pa.ancestry_table
print(at)

>      sample       left       right  ancestor  population
> 0         0        0.0     33027.0        74           0
> 1         0    33027.0    155453.0        33           0
> 2         0   155453.0    290542.0        46           0
> 3         0   290542.0    419848.0        18           0
> 4         0   419848.0    483009.0        83           1
> ..      ...        ...         ...       ...         ...
> 133       3  8672850.0   8849756.0        95           1
> 134       3  8849756.0   8918727.0       131           1
> 135       3  8918727.0   9165035.0        44           0
> 136       3  9165035.0   9176562.0        47           0
> 137       3  9176562.0  10000000.0        58           0
```

Both the *tspop.PopAncestry.squashed_table* and the *tspop.PopAncestry.ancestry_table* are pandas dataframes, so can be analysed using standard operations.

5.1 Example: calculating global ancestry

For instance, we could get the sum of all regions inherited from an ancestor in population 0 like this. We'll first subset the *tspop.PopAncestry.squashed_table* to only those tracts inherited from an ancestor in population 0:

```
st0 = st[st.population == 0]
print(st0)

>     sample       left       right  population
> 0        0        0.0    419848.0           0
> 2        0   483009.0   1475765.0           0
> 4        0  2427904.0   3635390.0           0
> 6        0  4606954.0   6277367.0           0
> ..     ...        ...         ...         ...         ...
> 52       3  7043989.0   7134130.0           0
> 54       3  7362300.0   7369409.0           0
> 56       3  7596783.0   8289015.0           0
> 58       3  8918727.0  10000000.0           0
```

By summing the tract lengths in the rows, we get the length of the tracts from population 0:

```
pop0_lengths = sum(st0.right - st0.left)
print(pop0_lengths)

> 23278398.0
```

Dividing this by the sum of the genomic lengths in the *tspop.PopAncestry* object gives the proportion of the genomes that were inherited from individuals in population 0, with reference to the ancestors present at the census time:

```
print(pop0_lengths/pa.total_genome_length)

> 0.58195995
```

CHAPTER

SIX

THE IDEAS BEHIND `TSPOP`

Simulated tree sequences contain richly detailed information about local ancestry: any sample node that descends from a node in a given population at some genomic location will have ancestry with the population at that location.

However, for realistically large and complicated simulations, it is difficult to recover this information from the overall genealogies. A visually intuitive way to do this is to locate each sample haplotype on each tree and trace a path up the tree until an ancestral node from one of the populations of interest is reached.

Unfortunately, this approach will be quite inefficient. Any genealogical feature that is shared between different haplotypes, or across different regions of the genome, will be processed separately for each sample and each tree. Given the substantial correlations in genealogy that typically exist between individuals, and across genomes, this approach would require many repetitive operations.

To extract local ancestry from a tree sequence, there are several essential steps:

1. Make a record of which nodes belong to ancestors in the populations of interest.
2. For each genomic segment belonging to a present-day sample, trace a path upwards through the trees to determine which of the nodes in the first step are ancestral to each subsegment.
3. Look up the population p that each ancestral node a belongs to. Then any segments that descend from a have local ancestry with population p.

Step 2 is potentially complicated and inefficient. This is the operation performed efficiently by the *link-ancestors* method in *tskit*. Essentially, *link-ancestors* performs a 'simplification' of the tree sequence so that relationships between samples and ancestors of interest are shown directly.

Note: Algorithmic details are in the preprint.

CHAPTER SEVEN

API

class tspop.**PopAncestry**(*left*, *right*, *population*, *ancestor*, *child*, *sample_nodes*, *sequence_length*)

Bases: `object`

In most cases, this should be created with the *`tspop.get_pop_ancestry()`* method. An object holding local ancestry information, and various summaries of that information.

Parameters

- **`left`** (*`list(float)`*) – The array of left coordinates.
- **`right`** (*`list(float)`*) – The array of right coordinates.
- **`population`** (*`list(int)`*) – The array of population labels.
- **`sample_nodes`** (*`list(int)`*) – The list of IDs corresponding to sample nodes.
- **`sequence_length`** (*`float`*) – The physical length of the region represented.

`ancestry_table`

A pandas.DataFrame object with column labels `sample`, `left`, `right`, `ancestor`, `population`. Each row (`sample`, `left`, `right`, `ancestor`, `population`) indicates that over the genomic interval with coordinates [`left`, `right`), the sample node with ID `sample` has inherited from the ancestral node with ID `ancestor` in the population with ID `population`. Ancestral nodes and population labels are taken from the specified census time.

`ancestry_table_write_csv`(*outfile*, ***kwargs*)

Writes the ancestry table to a csv file.

param outfile: The name of the output file. type: str param kwargs: other keyword arguments for pandas.to_csv

`calculate_ancestry_fraction`(*population*, *sample=None*)

Returns the total fraction of genomic material inherited from a given population.

Parameters

- **`population`** (*`int`*) – The index of the population to use.
- **`sample`** (*`int`*) – A specific sample node.

Returns

the global ancestry fraction.

`coverage`

The proportion of the total genome length with an ancestor in the *`tspop.PopAncestry.squashed_table`* and *`tspop.PopAncestry.ancestry_table`*.

`num_ancestors`

The number of ancestral haplotypes. Strictly less than or equal to the number of inputted ancestral nodes.

`num_samples`

The number of provided samples.

`plot_karyotypes`(*sample_pair*, *colors=None*, *pop_labels=None*, *title=None*, *length_in_Mb=True*, *outfile=None*, *height=12*, *width=20*)

Note: Diploid only for now.

Creates a plot of the ancestry tracts in a sample pair of chromosomes using `matplotlib`.

Parameters

- **`sample_pair`** (*`list(int)`*) – a pair of sample node IDs in the PopAncestry object.
- **`colors`** (*`list(str)`*) – A list of pyplot-compatible colours to use for the ancestral populations, given in order of their appearance in the *`tspop.PopAncestry.squashed_table`*. If None, uses the default matplotlib colour cycle.
- **`pop_labels`** (*`list(str)`*) – Ancestral population labels for the plot legend. If None, defaults to Pop0, Pop1 etc.
- **`title`** (*`str`*) – The title of the plot. If None, defaults to 'Ancestry in admixed individual'.
- **`length_in_Mb`** (*`bool`*) – Whether or not to label the horizontal axis in megabases. Defaults to True.
- **`outfile`** (*`str`*) – The name of the output file. If None, the plot opens with the system viewer.
- **`height`** (*`float`*) – The height of the figure in inches.
- **`width`** (*`float`*) – The width of the figure in inches.

Returns

a matplotlib figure.

`squashed_table`

A pandas.DataFrame object with column labels `sample`, `left`, `right`, `population`. Each row (`sample`, `left`, `right`, `population`) indicates that over the genomic interval with coordinates [`left`, `right`), the sample node with ID `sample` has inherited from an ancestral node in the population with ID `population`. Population labels are taken from the specified census time.

`squashed_table_write_csv`(*outfile*, ***kwargs*)

Writes the squashed table to a csv file.

param outfile: The name of the output file. type: str param kwargs: other keyword arguments for pandas.to_csv

`subset_tables`(*subset_samples*, *inplace=False*)

Subsets the ancestry table and squashed table by sample. Note: by default this returns a copy of the original tables. To overwrite the original tables, set inplace=True. (In this case, the function returns nothing).

Parameters

- **`subset_samples`** (*`list(int)`*) – The sample nodes to keep.
- **`inplace`** (*`bool`*) – Whether to overwrite the original tables.

Returns
The subsetted ancestry table and squashed table (only if *inplace=True*).

`total_genome_length`
Sequence length times the number of samples.

`tspop.get_pop_ancestry`(*ts*, *census_time*)
Creates a *tspop.PopAncestry* object from a simulated tree sequence containing ancestral census nodes. These are the ancestors that population-based ancestry will be calculated with respect to.

Parameters

- **ts** (*tskit.TreeSequence*) – A tree sequence containing census nodes.
- **census_time** (*list(int)*) – The time at which the census nodes are recorded.

Returns
a *tspop.PopAncestry* object

0.0.2: April 2023 - Added *subset_tables*, *ancestry_table_write_csv* and *squashed_table_write_csv*.

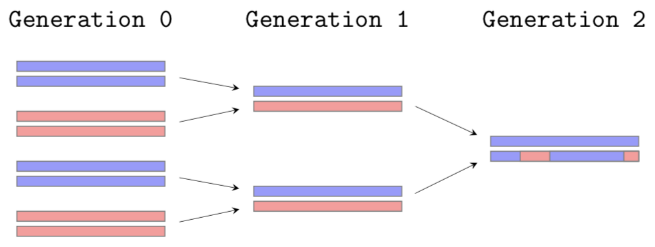
0.0.1: July 2022 - Initial release

CHAPTER

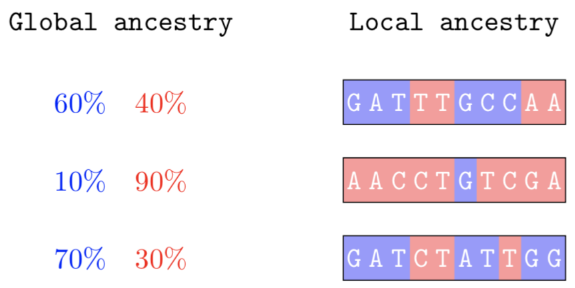
EIGHT

ABOUT TSPOP

Suppose your genealogical ancestors can be partitioned into distinct populations (represented here by different colours):



This is typically reported as global and local ancestry:



Using `msprime` and `SLiM`, you can simulate under detailed models of migration and population structure. This is the documentation for `tspop`, a lightweight package that makes it easier for you to extract information about population-based ancestry from these simulations.

Note: Add link to preprint/note when it's written.

Under the hood, `tspop` relies on

- the `tskit` package to efficiently extract the population-based information in the simulated datasets.
- the `pandas` package to provide user-friendly, interpretable output.

8.1 First steps

- Head to the *Installation* page to install `tspop` on your computer.
- Population-based ancestry is **not well-defined** without some notion of a *census time*. Read *Simulation setup* to see how to design your simulations to ensure they will work with `tspop`.
- Flick through the examples to see `tspop` in action.
- Check out *The ideas behind tspop* to learn more about why `tspop` is so efficient.

PYTHON MODULE INDEX

INDEX