msprime Documentation

Release 0.4.1.dev0+ng71b83cb.d20161007

Jerome Kelleher

October 07, 2016

Contents

This is the documentation for `msprime`, a reimplementation of Hudson's classical **`ms`** simulator.

CHAPTER 1

Contents:

1.1 Introduction

The primary goal of msprime is to efficiently and conveniently generate coalescent trees for a sample under a range of evolutionary scenarios. The library is a reimplementation of Hudson's seminal ms program, and aims to eventually reproduce all its functionality. msprime differs from ms in some important ways:

1. msprime is *much* more efficient than ms, both in terms of memory usage and simulation time. In fact, msprime is also much more efficient than simulators based on approximations to the coalescent with recombination model, especially for simulations with very large sample sizes. msprime can easily simulate chromosome sized regions for hundreds of thousands of samples.
2. msprime is primarily designed to be used through its *Python API* to simplify the workflow associated with running and analysing simulations. (However, we do provide an ms-compatible *command line interface* to plug in to existing workflows.) For many simulations we first write a script to generate the command line parameters we want to run, then fork shell processes to run the simulations, and then parse the results to obtain the genealogies in a form we can use. With msprime all of this can be done directly in Python, which is both simpler and far more efficient.
3. msprime does not use Newick trees for interchange as they are extremely inefficient in terms of the time required to generate and parse, as well as the space required to store them. Instead, we use a *well-defined* format using the powerful HDF5 standard. This format allows us to store genealogical data very concisely, particularly for large sample sizes.

1.2 Tutorial

This is the tutorial for the Python interface to the msprime library. Detailed *API Documentation* is also available for this library. An **ms**-compatible *command line interface* is also available if you wish to use msprime directly within an existing work flow.

1.2.1 Simulating trees

Running simulations is very straightforward in msprime:

```
>>> import msprime
>>> tree_sequence = msprime.simulate(sample_size=5, Ne=1000)
>>> tree = next(tree_sequence.trees())
>>> print(tree)
{0: 5, 1: 7, 2: 5, 3: 7, 4: 6, 5: 6, 6: 8, 7: 8, 8: -1}
```

Here, we simulate the coalescent for a sample of size 5 with an effective population size of 1000, and then print out a summary of the resulting tree. The *simulate()* function returns a *TreeSequence* object, which provides a very efficient way to access the correlated trees in simulations involving recombination. In this example we know that there can only be one tree because we have not provided a value for `recombination_rate`, and it defaults to zero. Therefore, we access the only tree in the sequence using the call `next(tree_sequence.trees())`.

Trees are represented within `msprime` in a slightly unusual way. In the majority of libraries dealing with trees, each node is represented as an object in memory and the relationship between nodes as pointers between these objects. In `msprime`, however, nodes are *integers*: the leaves (i.e., our sample) are the integers 0 to $n - 1$, and every internal node is some positive integer $\geq n$. The result of printing the tree is a summary of how these nodes relate to each other in terms of their parents. For example, we can see that the parent of nodes 1 and 3 is node 7.

This relationship can be seen more clearly in a picture:

This image shows the same tree as in the example but drawn out in a more familiar format (images like this can be drawn for any tree using the *draw()* method). We can see that the leaves of the tree are labelled with 0 to 4, and all the internal nodes of the tree are also integers with the root of the tree being 8. Also shown here are the times for each internal node in generations. (The time for all leaves is 0, and so we don't show this information to avoid clutter.)

Knowing that our leaves are 0 to 4, we can easily trace our path back to the root for a particular sample using the *get_parent()* method:

```
>>> u = 0
>>> while u != msprime.NULL_NODE:
>>>     print("node {}: time = {}".format(u, tree.get_time(u)))
>>>     u = tree.get_parent(u)
node 0: time = 0.0
node 5: time = 107.921165302
node 6: time = 1006.74711128
node 8: time = 1785.36352521
```

In this code chunk we iterate up the tree starting at node 0 and stop when we get to the root. We know that a node is the root if its parent is `msprime.NULL_NODE`, which is a special reserved node. (The value of the null node is -1, but we recommend using the symbolic constant to make code more readable.) We also use the *get_time()* method to get the time for each node, which corresponds to the time in generations at which the coalescence event happened during the simulation. We can also obtain the length of a branch joining a node to its parent using the *get_branch_length()* method:

```
>>> print(tree.get_branch_length(6))
778.616413923
```

The branch length for node 6 is 778.6 generations as the time for node 6 is 1006.7 and the time of its parent is 1785.4. It is also often useful to obtain the total branch length of the tree, i.e., the sum of the lengths of all branches:

```
>>> print(tree.get_total_branch_length())
>>> 5932.15093686
```

1.2.2 Recombination

Simulating the history of a single locus is a very useful, but we are most often interesting in simulating the history of our sample across large genomic regions under the influence of recombination. The `msprime` API is specifically designed to make this common requirement both easy and efficient. To model genomic sequences under the influence of recombination we have two parameters to the *simulate()* function. The `length` parameter specifies the length of the simulated sequence in bases, and may be a floating point number. If `length` is not supplied, it is assumed to be 1. The `recombination_rate` parameter specifies the rate of crossing over per base per generation, and is zero by default. See the *API Documentation* for a discussion of the precise recombination model used.

Here, we simulate the trees across over a 10kb region with a recombination rate of 2×10^{-8} per base per generation, with an effective population size of 1000:

```
>>> tree_sequence = msprime.simulate(
...     sample_size=5, Ne=1000, length=1e4, recombination_rate=2e-8)
>>> for tree in tree_sequence.trees():
...     print(tree.get_interval(), str(tree), sep="\t")
(0.0, 4701.4225005874)      {0: 6, 1: 5, 2: 6, 3: 9, 4: 5, 5: 7, 6: 7, 7: 9, 9: -1}
(4701.4225005874, 10000.0)  {0: 6, 1: 5, 2: 6, 3: 8, 4: 5, 5: 8, 6: 9, 8: 9, 9: -1}
```

In this example, we use the *trees()* method to iterate over the trees in the sequence. For each tree we print out the interval the tree covers (i.e., the genomic coordinates which all share precisely this tree) using the *get_interval()* method. Thus, the first tree covers the first 4.7kb of sequence and the second tree covers the remaining 5.3kb. We also print out the summary of each tree in terms of the parent values for each tree. Again, these differences are best illustrated by some images:

(We have suppressed the node time labels here for clarity.) We can see that these trees share a great deal of their structure, but that there are also important differences between the trees.

Warning: Do not store the values returned from the *trees()* iterator in a list and operate on them afterwards! For efficiency reasons `msprime` uses the same instance of *SparseTree* for each tree in the sequence and updates the internal state for each new tree. Therefore, if you store the trees returned from the iterator in a list, they will all refer to the same tree.

1.2.3 Mutations

Mutations are generated in `msprime` by throwing mutations down on the branches of trees at a particular rate. The mutations are generated under the infinite sites model, and so each mutation occurs at a unique (floating point) point position along the genomic interval occupied by a tree. The mutation rate for simulations is specified using the `mutation_rate` parameter of *simulate()*. For example, to add some mutations to our example above, we can use:

```
>>> tree_sequence = msprime.simulate(
...     sample_size=5, Ne=1000, length=1e4, recombination_rate=2e-8, mutation_rate=2e-8)
>>> print("Total mutations = ", tree_sequence.get_num_mutations())
>>> for tree in tree_sequence.trees():
>>>     print(tree.get_interval(), list(tree.mutations()), sep="\t")
Total mutations =  1
(0.0, 4701.4225005874)  []
(4701.4225005874, 10000.0)      [Mutation(position=5461.212369738915, node=6, index=0)]
```

In this example (which has the same genealogies as our example above because we use the same random seed), we have one mutation which falls on the second tree. Mutations are represented as an object with three attributes: `position` is the location of the mutation in genomic coordinates, `node` is the node in the tree above which the mutation occurs, and `index` is the (zero-based) index of the mutation in the list. Positions are given as a floating point value as we are using the infinite sites model. Every mutation falls on exactly one tree and we obtain the mutations for a particular tree using the *mutations()* method. Mutations are always returned in increasing order of position. The mutation in this example is shown as a red box on the corresponding branch:

We can calculate the allele frequency of mutations easily and efficiently using the *get_num_leaves()* which returns the number of leaves underneath a particular node. For example,:

```
>>> for tree in tree_sequence.trees():
...     for position, node in tree.mutations():
...         print("Mutation @ position {} has frequency {}".format(
...             mutation.position,
```

```
...             tree.get_num_leaves(mutation.node) / tree.get_sample_size()))
Mutation @ position 5461.21236974 has frequency 0.4
```

Sometimes we are only interested in a subset of the mutations in a tree sequence. In these situations, it is useful (and efficient) to update the tree sequence to only include the mutations we are interested in using the *TreeSequence.set_mutations()* method. Here, for example, we simulate some data and then retain only the common variants where the allele frequency is greater than 0.5.

```
import msprime

def set_mutations_example():
    tree_sequence = msprime.simulate(
        sample_size=10000, Ne=1e4, length=1e7, recombination_rate=2e-8,
        mutation_rate=2e-8)
    print("Simulated ", tree_sequence.get_num_mutations(), "mutations")
    common_mutations = []
    for tree in tree_sequence.trees():
        for mutation in tree.mutations():
            p = tree.get_num_leaves(mutation.node) / tree.get_sample_size()
            if p >= 0.5:
                common_mutations.append(mutation)
    tree_sequence.set_mutations(common_mutations)
    print("Reduced to ", tree_sequence.get_num_mutations(), "common mutations")
```

Running this code, we get:

```
>>> set_mutations_example()
Simulated  78202 mutations
Reduced to  5571 common mutations
```

1.2.4 Variants

We are often interesting in accessing the sequence data that results from simulations directly. The most efficient way to do this is by using the *TreeSequence.variants()* method, which returns an iterator over all the variant objects arising from the trees and mutations. Each variant contains all the information in a mutation object, but also has the observed sequences for each sample in the `genotypes` field.

```
import msprime

def variants_example():
    tree_sequence = msprime.simulate(
        sample_size=20, Ne=1e4, length=5e3, recombination_rate=2e-8,
        mutation_rate=2e-8, random_seed=10)
    print("Simulated ", tree_sequence.get_num_mutations(), "mutations")
    for variant in tree_sequence.variants():
        print(variant.index, variant.position, variant.genotypes, sep="\t")
```

In this example we simulate some data and then print out the observed sequences. We loop through each variant and print out the observed state of each sample as an array of zeros and ones, along with the index and position of the corresponding mutation. (The default form for the `genotypes` array here is a `numpy.ndarray`; however, the output can also be a plain Python bytes object. See the *TreeSequence.variants()* documentation for details.) Running the code, we get:

```
>>> variants_example()
Simulated  7 mutations
0       2146.29801511   [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
1       2475.24314909   [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
```

```
2       3087.04505359   [0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
3       3628.35359621   [1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1]
4       4587.85827679   [0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 0 0 0 0 0]
5       4593.29453791   [1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1]
6       4784.26662856   [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
```

This way of working with the sequence data is quite efficient because we do not need to keep the entire variant matrix in memory at once.

```
import msprime
import numpy as np

def variant_matrix_example():
    print("\nCreating full variant matrix")
    tree_sequence = msprime.simulate(
        sample_size=20, Ne=1e4, length=5e3, recombination_rate=2e-8,
        mutation_rate=2e-8, random_seed=10)
    shape = tree_sequence.get_num_mutations(), tree_sequence.get_sample_size()
    A = np.empty(shape, dtype="u1")
    for variant in tree_sequence.variants():
        A[variant.index] = variant.genotypes
    print(A)
```

In this example, we run the same simulation but this time store entire variant matrix in a two-dimensional numpy array. This is useful for integrating with tools such as scikit allel.:

```
>>> variant_matrix_example()
Creating full variant matrix
[[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 0 0 0 0 0]
 [1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]]
```

1.2.5 Historical samples

Simulating coalescent histories in which some of the samples are not from the present time is straightforward in `msprime`. By using the `samples` argument to *`msprime.simulate()`* we can specify the location and time at which all samples are made.

```
def historical_samples_example():
    samples = [
        msprime.Sample(population=0, time=0),
        msprime.Sample(0, 0),  # Or, we can use positional arguments.
        msprime.Sample(0, 3.0)
    ]
    tree_seq = msprime.simulate(samples=samples)
    tree = next(tree_seq.trees())
    for u in range(tree_seq.get_num_nodes()):
        print(u, tree.get_parent(u), tree.get_time(u), sep="\t")
```

In this example we create three samples, two taken at the present time and one taken 1.0 generations in the past. There are a number of different ways in which we can describe the samples using the `msprime.Sample` object (samples can be provided as plain tuples also if more convenient). Running this example, we get:

```
>>> historical_samples_example()
0       3       0.0
1       3       0.0
2       4       1.0
3       4       0.502039955384
4       -1      4.5595966593
```

Because nodes 0 and 1 were sampled at time 0, their times in the tree are both 0. Node 2 was sampled at time 1.0, and so its time is recorded as 1.0 in the tree.

1.2.6 Replication

A common task for coalescent simulations is to check the accuracy of analytical approximations to statistics of interest. To do this, we require many independent replicates of a given simulation. msprime provides a simple and efficient API for replication: by providing the num_replicates argument to the *simulate()* function, we can iterate over the replicates in a straightforward manner. Here is an example where we compare the analytical results for the number of segregating sites with simulations:

```
import msprime
import numpy as np

def segregating_sites_example(n, theta, num_replicates):
    S = np.zeros(num_replicates)
    replicates = msprime.simulate(
        sample_size=n,
        mutation_rate=theta / 4,
        num_replicates=num_replicates)
    for j, tree_sequence in enumerate(replicates):
        S[j] = tree_sequence.get_num_mutations()
    # Now, calculate the analytical predictions
    S_mean_a = np.sum(1 / np.arange(1, n)) * theta
    S_var_a = (
        theta * np.sum(1 / np.arange(1, n)) +
        theta**2 * np.sum(1 / np.arange(1, n)**2))
    print("              mean              variance")
    print("Observed      {}\t\t{}".format(np.mean(S), np.var(S)))
    print("Analytical    {:.5f}\t\t{:.5f}".format(S_mean_a, S_var_a))
```

Running this code, we get:

```
>>> segregating_sites_example(10, 5, 100000)
              mean                  variance
Observed      14.12173              52.4695318071
Analytical    14.14484              52.63903
```

Note that in this example we did not provide a value for the Ne argument to *simulate()*. In this case the effective population size defaults to 1, which can be useful for theoretical work. However, it is essential to remember that all rates and times must still be scaled by 4 to convert into the coalescent time scale.

1.2.7 Population structure

Population structure in msprime closely follows the model used in the ms simulator: we have N demes with an $N \times N$ matrix describing the migration rates between these subpopulations. The sample sizes, population sizes and growth rates of all demes can be specified independently. Migration rates are specified using a migration matrix. Unlike ms however, all times and rates are specified in generations and all populations sizes are absolute (that is, not multiples of N_e).

In the following example, we calculate the mean coalescence time for a pair of lineages sampled in different demes in a symmetric island model, and compare this with the analytical expectation.

```
import msprime
import numpy as np

def migration_example():
    # M is the overall symmetric migration rate, and d is the number
    # of demes.
    M = 0.2
    d = 3
    # We rescale m into per-generation values for msprime.
    m = M / (4 * (d - 1))
    # Allocate the initial sample. Because we are interested in the
    # between deme coalescence times, we choose one sample each
    # from the first two demes.
    population_configurations = [
        msprime.PopulationConfiguration(sample_size=1),
        msprime.PopulationConfiguration(sample_size=1),
        msprime.PopulationConfiguration(sample_size=0)]
    # Now we set up the migration matrix. Since this is a symmetric
    # island model, we have the same rate of migration between all
    # pairs of demes. Diagonal elements must be zero.
    migration_matrix = [
        [0, m, m],
        [m, 0, m],
        [m, m, 0]]
    # We pass these values to the simulate function, and ask it
    # to run the required number of replicates.
    num_replicates = 1e6
    replicates = msprime.simulate(
        population_configurations=population_configurations,
        migration_matrix=migration_matrix,
        num_replicates=num_replicates)
    # And then iterate over these replicates
    T = np.zeros(num_replicates)
    for i, tree_sequence in enumerate(replicates):
        tree = next(tree_sequence.trees())
        # Convert the TMRCA to coalecent units.
        T[i] = tree.get_time(tree.get_root()) / 4
    # Finally, calculate the analytical expectation and print
    # out the results
    analytical = d / 2 + (d - 1) / (2 * M)
    print("Observed  =", np.mean(T))
    print("Predicted =", analytical)
```

Running this example we get:

```
>>> migration_example()
Observed  = 6.50638181614
Predicted = 6.5
```

1.2.8 Demography

Msprime provides a flexible and simple way to model past demographic events in arbitrary combinations. Here is an example describing the Gutenkunst et al. out-of-Africa model. See Figure 2B for a schematic of this model, and Table 1 for the values used.

Todo

Add a diagram of the model for convenience.

```
def out_of_africa():
    # First we set out the maximum likelihood values of the various parameters
    # given in Table 1.
    N_A = 7300
    N_B = 2100
    N_AF = 12300
    N_EU0 = 1000
    N_AS0 = 510
    # Times are provided in years, so we convert into generations.
    generation_time = 25
    T_AF = 220e3 / generation_time
    T_B = 140e3 / generation_time
    T_EU_AS = 21.2e3 / generation_time
    # We need to work out the starting (diploid) population sizes based on
    # the growth rates provided for these two populations
    r_EU = 0.004
    r_AS = 0.0055
    N_EU = N_EU0 / math.exp(-r_EU * T_EU_AS)
    N_AS = N_AS0 / math.exp(-r_AS * T_EU_AS)
    # Migration rates during the various epochs.
    m_AF_B = 25e-5
    m_AF_EU = 3e-5
    m_AF_AS = 1.9e-5
    m_EU_AS = 9.6e-5
    # Population IDs correspond to their indexes in the population
    # configuration array. Therefore, we have 0=YRI, 1=CEU and 2=CHB
    # initially.
    population_configurations = [
        msprime.PopulationConfiguration(
            sample_size=0, initial_size=N_AF),
        msprime.PopulationConfiguration(
            sample_size=1, initial_size=N_EU, growth_rate=r_EU),
        msprime.PopulationConfiguration(
            sample_size=1, initial_size=N_AS, growth_rate=r_AS)
    ]
    migration_matrix = [
        [      0, m_AF_EU, m_AF_AS],
        [m_AF_EU,       0, m_EU_AS],
        [m_AF_AS, m_EU_AS,       0],
    ]
    demographic_events = [
        # CEU and CHB merge into B with rate changes at T_EU_AS
        msprime.MassMigration(
            time=T_EU_AS, source=2, destination=1, proportion=1.0),
        msprime.MigrationRateChange(time=T_EU_AS, rate=0),
        msprime.MigrationRateChange(
            time=T_EU_AS, rate=m_AF_B, matrix_index=(0, 1)),
        msprime.MigrationRateChange(
            time=T_EU_AS, rate=m_AF_B, matrix_index=(1, 0)),
        msprime.PopulationParametersChange(
            time=T_EU_AS, initial_size=N_B, growth_rate=0, population_id=1),
        # Population B merges into YRI at T_B
        msprime.MassMigration(
```

```
            time=T_B, source=1, destination=0, proportion=1.0),
        # Size changes to N_A at T_AF
        msprime.PopulationParametersChange(
            time=T_AF, initial_size=N_A, population_id=0)
    ]
    # Use the demography debugger to print out the demographic history
    # that we have just described.
    dp = msprime.DemographyDebugger(
        Ne=N_A,
        population_configurations=population_configurations,
        migration_matrix=migration_matrix,
        demographic_events=demographic_events)
    dp.print_history()
```

The *DemographyDebugger* provides a method to debug the history that you have described so that you can be sure that the migration rates, population sizes and growth rates are all as you intend during each epoch:

```
=============================
Epoch: 0 -- 848.0 generations
=============================
     start     end      growth_rate |     0        1        2
   -------- --------       -------- | -------- -------- --------
0 |1.23e+04 1.23e+04              0 |     0      3e-05   1.9e-05
1 |2.97e+04   1e+03           0.004 |   3e-05      0     9.6e-05
2 |5.41e+04    510           0.0055 |  1.9e-05  9.6e-05     0

Events @ generation 848.0
   - Mass migration: lineages move from 2 to 1 with probability 1.0
   - Migration rate change to 0 everywhere
   - Migration rate change for (0, 1) to 0.00025
   - Migration rate change for (1, 0) to 0.00025
   - Population parameter change for 1: initial_size -> 2100 growth_rate -> 0

================================
Epoch: 848.0 -- 5600.0 generations
================================
     start     end      growth_rate |     0        1        2
   -------- --------       -------- | -------- -------- --------
0 |1.23e+04 1.23e+04              0 |     0     0.00025     0
1 | 2.1e+03  2.1e+03              0 |  0.00025     0        0
2 |5.41e+04 2.41e-07         0.0055 |     0        0        0

Events @ generation 5600.0
   - Mass migration: lineages move from 1 to 0 with probability 1.0

================================
Epoch: 5600.0 -- 8800.0 generations
================================
     start     end      growth_rate |     0        1        2
   -------- --------       -------- | -------- -------- --------
0 |1.23e+04 1.23e+04              0 |     0     0.00025     0
1 | 2.1e+03  2.1e+03              0 |  0.00025     0        0
2 |5.41e+04  0.00123          0.0055 |     0        0        0

Events @ generation 8800.0
   - Population parameter change for 0: initial_size -> 7300

================================
Epoch: 8800.0 -- inf generations
```

```
================================
     start      end      growth_rate |      0         1         2
   -------- --------       -------- | -------- -------- --------
0 | 7.3e+03  7.3e+03              0 |      0     0.00025     0
1 | 2.1e+03  2.1e+03              0 |  0.00025     0         0
2 |5.41e+04     0            0.0055 |      0         0         0
```

Warning: The output of the *DemographyDebugger.print_history()* method is intended only for debugging purposes, and is not meant to be machine readable. The format is also preliminary; if there is other information that you think would be useful, please open an issue on GitHub

Once you are satisfied that the demographic history that you have built is correct, it can then be simulated by calling the *simulate()* function.

1.2.9 Recombination maps

The `msprime` API allows us to quickly and easily simulate data from an arbitrary recombination map. In this example we read a recombination map for human chromosome 22, and simulate a single replicate. After the simulation is completed, we plot histograms of the recombination rates and the simulated breakpoints. These show that density of breakpoints follows the recombination rate closely.

```
import numpy as np
import scipy.stats
import matplotlib.pyplot as pyplot

def variable_recomb_example():
    infile = "hapmap/genetic_map_GRCh37_chr22.txt"
    # Read in the recombination map using the read_hapmap method,
    recomb_map = msprime.RecombinationMap.read_hapmap(infile)

    # Now we get the positions and rates from the recombination
    # map and plot these using 500 bins.
    positions = np.array(recomb_map.get_positions()[1:])
    rates = np.array(recomb_map.get_rates()[1:])
    num_bins = 500
    v, bin_edges, _ = scipy.stats.binned_statistic(
        positions, rates, bins=num_bins)
    x = bin_edges[:-1][np.logical_not(np.isnan(v))]
    y = v[np.logical_not(np.isnan(v))]
    fig, ax1 = pyplot.subplots(figsize=(16, 6))
    ax1.plot(x, y, color="blue")
    ax1.set_ylabel("Recombination rate")
    ax1.set_xlabel("Chromosome position")

    # Now we run the simulation for this map. We assume Ne=10^4
    # and have a sample of 100 individuals
    tree_sequence = msprime.simulate(
        sample_size=100,
        Ne=10**4,
        recombination_map=recomb_map)
    # Now plot the density of breakpoints along the chromosome
    breakpoints = np.array(list(tree_sequence.breakpoints()))
    ax2 = ax1.twinx()
    v, bin_edges = np.histogram(breakpoints, num_bins, density=True)
    ax2.plot(bin_edges[:-1], v, color="green")
    ax2.set_ylabel("Breakpoint density")
```

```
    ax2.set_xlim(1.5e7, 5.3e7)
    fig.savefig("hapmap_chr22.svg")
```

1.2.10 Calculating LD

The `msprime` API provides methods to efficiently calculate population genetics statistics. For example, the *`LdCalculator`* class allows us to compute pairwise linkage disequilibrium coefficients. Here we use the *`get_r2_matrix()`* method to easily make an LD plot using matplotlib. (Thanks to the excellent scikit-allel for the basic plotting code used here.)

```
import msprime
import matplotlib.pyplot as pyplot

def ld_matrix_example():
    ts = msprime.simulate(100, recombination_rate=10, mutation_rate=20,
            random_seed=1)
    ld_calc = msprime.LdCalculator(ts)
    A = ld_calc.get_r2_matrix()
    # Now plot this matrix.
    x = A.shape[0] / pyplot.rcParams['savefig.dpi']
    x = max(x, pyplot.rcParams['figure.figsize'][0])
    fig, ax = pyplot.subplots(figsize=(x, x))
    fig.tight_layout(pad=0)
    im = ax.imshow(A, interpolation="none", vmin=0, vmax=1, cmap="Blues")
    ax.set_xticks([])
    ax.set_yticks([])
    for s in 'top', 'bottom', 'left', 'right':
        ax.spines[s].set_visible(False)
    pyplot.gcf().colorbar(im, shrink=.5, pad=0)
    pyplot.savefig("ld.svg")
```

1.2.11 Working with threads

When performing large calculations it's often useful to split the work over multiple processes or threads. The msprime API can be used without issues across multiple processes, and the Python `multiprocessing` module often provides a very effective way to work with many replicate simulations in parallel.

When we wish to work with a single very large dataset, however, threads can offer better resource usage because of the shared memory space. The Python `threading` library gives a very simple interface to lightweight CPU threads and allows us to perform several CPU intensive tasks in parallel. The `msprime` API is designed to allow multiple threads to work in parallel when CPU intensive tasks are being undertaken.

Note: In the CPython implementation the Global Interpreter Lock ensures that only one thread executes Python bytecode at one time. This means that Python code does not parallelise well across threads, but avoids a large number of nasty pitfalls associated with multiple threads updating data structures in parallel. Native C extensions like `numpy` and `msprime` release the GIL while expensive tasks are being performed, therefore allowing these calculations to proceed in parallel.

In the following example we wish to find all mutations that are in approximate LD ($r^2 \geq 0.5$) with a given set of mutations. We parallelise this by splitting the input array between a number of threads, and use the *`LdCalculator.get_r2_array()`* method to compute the r^2 value both up and downstream of each focal mutation, filter out those that exceed our threshold, and store the results in a dictionary. We also use the very cool tqdm module to give us a progress bar on this computation.

```
import threading
import numpy as np
import tqdm
import msprime

def find_ld_sites(
        tree_sequence, focal_mutations, max_distance=1e6, r2_threshold=0.5,
        num_threads=8):
    results = {}
    progress_bar = tqdm.tqdm(total=len(focal_mutations))
    num_threads = min(num_threads, len(focal_mutations))

    def thread_worker(thread_index):
        ld_calc = msprime.LdCalculator(tree_sequence)
        chunk_size = int(math.ceil(len(focal_mutations) / num_threads))
        start = thread_index * chunk_size
        for focal_mutation in focal_mutations[start: start + chunk_size]:
            a = ld_calc.get_r2_array(
                focal_mutation, max_distance=max_distance,
                direction=msprime.REVERSE)
            rev_indexes = focal_mutation - np.nonzero(a >= r2_threshold)[0] - 1
            a = ld_calc.get_r2_array(
                focal_mutation, max_distance=max_distance,
                direction=msprime.FORWARD)
            fwd_indexes = focal_mutation + np.nonzero(a >= r2_threshold)[0] + 1
            indexes = np.concatenate((rev_indexes[::-1], fwd_indexes))
            results[focal_mutation] = indexes
            progress_bar.update()

    threads = [
        threading.Thread(target=thread_worker, args=(j,))
        for j in range(num_threads)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    progress_bar.close()
    return results

def threads_example():
    ts = msprime.simulate(
        sample_size=1000, Ne=1e4, length=1e7, recombination_rate=2e-8,
        mutation_rate=2e-8)
    counts = np.zeros(ts.get_num_mutations())
    for t in ts.trees():
        for mutation in t.mutations():
            counts[mutation.index] = t.get_num_leaves(mutation.node)
    doubletons = np.nonzero(counts == 2)[0]
    results = find_ld_sites(ts, doubletons, num_threads=8)
    print(
        "Found LD sites for", len(results), "doubleton mutations out of",
        ts.get_num_mutations())
```

In this example, we first simulate 1000 samples of 10 megabases and find all doubleton mutations in the resulting tree sequence. We then call the `find_ld_sites()` function to find all mutations that are within 1 megabase of these doubletons and have an r^2 statistic of greater than 0.5.

The `find_ld_sites()` function performs these calculations in parallel using 8 threads. The real work is done

in the nested `thread_worker()` function, which is called once by each thread. In the thread worker, we first allocate an instance of the *`LdCalculator`* class. (It is **critically important** that each thread has its own instance of *`LdCalculator`*, as the threads will not work efficiently otherwise.) After this, each thread works out the slice of the input array that it is responsible for, and then iterates over each focal mutation in turn. After the r^2 values have been calculated, we then find the indexes of the mutations corresponding to values greater than 0.5 using `numpy.nonzero()`. Finally, the thread stores the resulting array of mutation indexes in the `results` dictionary, and moves on to the next focal mutation.

Running this example we get:

```
>>> threads_example()
100%|| 4045/4045 [00:09<00:00, 440.29it/s]
Found LD sites for 4045 doubleton mutations out of 60100
```

1.3 API Documentation

This is the API documentation for `msprime`, and provides detailed information on the Python programming interface. See the *Tutorial* for an introduction to using this API to run simulations and analyse the results.

1.3.1 Simulation model

The simulation model in `msprime` closely follows the classical `ms` program. Unlike `ms`, however, time is measured in generations rather than "coalescent units". Internally the same simulation algorithm is used, but `msprime` provides a translation layer to allow the user input times and rates in generations. Similarly, the times associated with the trees produced by `msprime` are in measured generations. To enable this translation from generations into coalescent units and vice-versa, a reference effective population size must be provided, which is given by the `Ne` parameter in the *`simulate()`* function. (Note that we assume diploid population sizes thoughout, since we scale by $4N_e$.) Population sizes for individual demes and for past demographic events are defined as absolute values, **not** scaled by `Ne`. All migration rates and growth rates are also per generation.

When running simulations we define the length in bases L of the sequence in question using the `length` parameter. This defines the coordinate space within which trees and mutations are defined. L is a continuous value, and coordinates can take any value from 0 to L. Mutations occur in an infinite sites process along this sequence, and mutation rates are specified per generation, per unit of sequence length. Thus, given the per-generation mutation rate μ, the rate of mutation over the entire sequence in coalescent time units is $\theta = 4N_e\mu L$. It is important to remember these scaling factors when comparing with analytical results!

Similarly, recombination rates are per base, per generation in `msprime`. Thus, given the per generation crossover rate r, the overall rate of recombination between the ends of the sequence in coalescent time units is $\rho = 4N_e r L$. Recombination events occur in a continuous coordinate space, so that breakpoints do not necessarily occur at integer locations. However, the underlying recombination model is finite, and the behaviour of a small number of loci can be modelled using the `RecombinationMap` class. However, this is considered an advanced feature and the majority of cases should be well served with the default recombination model and number of loci.

Population structure is modelled by specifying a fixed number of demes d, and a $d \times d$ matrix M of per generation migration rates. Each element of the matrix $M_{j,k}$ defines the fraction of population j that consists of migrants from population k in each generation. Each deme has an initial absolute population size s and a per generation exponential growth rate α. The size of a given population at time t in the past (measured in generations) is therefore given by $se^{-\alpha t}$. Demographic events that occur in the history of the simulated population alter some aspect of this population configuration at a particular time in the past.

Warning: This parameterisation of recombination, mutation and migration rates is different to **ms**, which states these rates over the entire region and in coalescent time units. The motivation for this is to allow the user change the size of the simulated region without having to rescale the recombination and mutation rates, and to also allow users directly state times and rates in units of generations. However, the `mspms` command line application is fully **ms** compatible.

1.3.2 Running simulations

The *simulate()* function provides the primary interface to running coalescent simulations in msprime.

msprime.**simulate**(*sample_size=None*, *Ne=1*, *length=None*, *recombination_rate=None*, *recombination_map=None*, *mutation_rate=None*, *population_configurations=None*, *migration_matrix=None*, *demographic_events=[]*, *samples=None*, *random_seed=None*, *num_replicates=None*)

Simulates the coalescent with recombination under the specified model parameters and returns the resulting *TreeSequence*.

Parameters

- **sample_size** (*int*) – The number of individuals in our sample. If not specified or None, this defaults to the sum of the subpopulation sample sizes. Either `sample_size`, `population_configurations` or `samples` must be specified.
- **Ne** (*float*) – The effective (diploid) population size for the reference population. This determines the factor by which the per-generation recombination and mutation rates are scaled in the simulation. This defaults to 1 if not specified.
- **length** (*float*) – The length of the simulated region in bases. This parameter cannot be used along with `recombination_map`. Defaults to 1 if not specified.
- **recombination_rate** (*float*) – The rate of recombination per base per generation. This parameter cannot be used along with `recombination_map`. Defaults to 0 if not specified.
- **recombination_map** (*RecombinationMap*) – The map describing the changing rates of recombination along the simulated chromosome. This parameter cannot be used along with the `recombination_rate` or `length` parameters, as these values are encoded within the map. Defaults to a uniform rate as described in the `recombination_rate` parameter if not specified.
- **mutation_rate** (*float*) – The rate of mutation per base per generation. If not specified, no mutations are generated.
- **population_configurations** (*list or None.*) – The list of *PopulationConfiguration* instances describing the sampling configuration, relative sizes and growth rates of the populations to be simulated. If this is not specified, a single population with a sample of size `sample_size` is assumed.
- **migration_matrix** (*list*) – The matrix describing the rates of migration between all pairs of populations. If N populations are defined in the `population_configurations` parameter, then the migration matrix must be an $N \times N$ matrix consisting of N lists of length N.
- **demographic_events** (*list*) – The list of demographic events to simulate. Demographic events describe changes to the populations in the past. Events should be supplied in non-decreasing order of time. Events with the same time value will be applied sequentially in the order that they were supplied before the simulation algorithm continues with the next time step.

- **samples** (*list*) – The list specifying the location and time of all samples. This parameter may be used to specify historical samples, and cannot be used in conjunction with the `sample_size` parameter. Each sample is a (`population_id`, `time`) pair such that the sample in position `j` in the list of samples is drawn in the specified population at the specfied time. Time is measured in generations, as elsewhere.
- **random_seed** (*int*) – The random seed. If this is *None*, a random seed will be automatically generated. Valid random seeds must be between 1 and $2^{32} - 1$.
- **num_replicates** (*int*) – The number of replicates of the specified parameters to simulate. If this is not specified or None, no replication is performed and a *TreeSequence* object returned. If `num_replicates` is provided, the specified number of replicates is performed, and an iterator over the resulting *TreeSequence* objects returned.

Returns The *TreeSequence* object representing the results of the simulation if no replication is performed, or an iterator over the independent replicates simulated if the `num_replicates` parameter has been used.

Return type *TreeSequence* or an iterator over *TreeSequence* replicates.

Warning If using replication, do not store the results of the iterator in a list! For performance reasons, the same underlying object may be used for every TreeSequence returned which will most likely lead to unexpected behaviour.

Population structure

Population structure is modelled in `msprime` by specifying a fixed number of demes, with the migration rates between those demes defined by a migration matrix. Each deme has an `initial_size` that defines its absolute size at time zero and a per-generation `growth_rate` which specifies the exponential growth rate of the sub-population. We must also define the size of the sample to draw from each deme. The number of populations and their initial configuration is defined using the `population_configurations` parameter to *simulate()*, which takes a list of *PopulationConfiguration* instances. Population IDs are zero indexed, and correspond to their position in the list.

Samples are drawn sequentially from populations in increasing order of population ID. For example, if we specified an overall sample size of 5, and specify that 2 samples are drawn from population 0 and 3 from population 1, then individuals 0 and 1 will be initially located in population 0, and individuals 2, 3 and 4 will be drawn from population 2.

Given N populations, migration matrices are specified using an $N \times N$ matrix of deme-to-deme migration rates. See the documentation for *simulate()* and the *Simulation model* section for more details on the migration rates.

class `msprime.`**PopulationConfiguration**(*sample_size=None*, *initial_size=None*, *growth_rate=0.0*)

The initial configuration of a population (or deme) in a simulation.

Parameters

- **sample_size** (*int*) – The number of initial samples that are drawn from this population.
- **initial_size** (*float*) – The absolute size of the population at time zero. Defaults to the reference population size N_e.
- **growth_rate** (*float*) – The exponential growth rate of the population per generation. Growth rates can be negative. This is zero for a constant population size. Defaults to 0.

Demographic Events

Demographic events change some aspect of the population configuration at some time in the past, and are specified using the `demographic_events` parameter to *`simulate()`*. Each element of this list must be an instance of one of the following demographic events that are currently supported. Note that all times are measured in generations, all sizes are absolute (i.e., *not* relative to N_e), and all rates are per-generation.

class `msprime.`**`PopulationParametersChange`**(*time*, *initial_size=None*, *growth_rate=None*, *population_id=None*)

Changes the demographic parameters of a population at a given time.

This event generalises the `-eg`, `-eG`, `-en` and `-eN` options from `ms`. Note that unlike `ms` we do not automatically set growth rates to zero when the population size is changed.

Parameters

- **`time`** (*`float`*) – The time at which this event occurs in generations.
- **`initial_size`** (*`float`*) – The absolute size of the population at the beginning of the time slice starting at `time`. If None, this is calculated according to the initial population size and growth rate over the preceding time slice.
- **`growth_rate`** (*`float`*) – The new per-generation growth rate. If None, the growth rate is not changed. Defaults to None.
- **`population_id`** (*`int`*) – The ID of the population affected. If `population_id` is None, the changes affect all populations simultaneously.

class `msprime.`**`MigrationRateChange`**(*time*, *rate*, *matrix_index=None*)

Changes the rate of migration to a new value at a specific time.

Parameters

- **`time`** (*`float`*) – The time at which this event occurs in generations.
- **`rate`** (*`float`*) – The new per-generation migration rate.
- **`matrix_index`** (*`tuple`*) – A tuple of two population IDs descibing the matrix index of interest. If `matrix_index` is None, all non-diagonal entries of the migration matrix are changed simultaneously.

class `msprime.`**`MassMigration`**(*time*, *source*, *destination*, *proportion=1.0*)

A mass migration event in which some fraction of the population in one deme simultaneously move to another deme, viewed backwards in time. For each lineage currently present in the source population, they move to the destination population with probability equal to `proportion`.

This event class generalises the population split (`-ej`) and admixture (`-es`) events from `ms`. Note that MassMigrations do *not* have any side effects on the migration matrix.

Parameters

- **`time`** (*`float`*) – The time at which this event occurs in generations.
- **`source`** (*`int`*) – The ID of the source population.
- **`destination`** (*`int`*) – The ID of the destination population.
- **`proportion`** (*`float`*) – The probability that any given lineage within the source population migrates to the destination population.

Debugging demographic models

Warning: The `DemographyDebugger` class is preliminary, and the API is likely to change in the future.

class msprime.**DemographyDebugger**(*Ne=1*, *population_configurations=None*, *migration_matrix=None*, *demographic_events=[]*)

A class to facilitate debugging of population parameters and migration rates in the past.

print_history(*output=<open file '<stdout>', mode 'w'>*)

Prints a summary of the history of the populations.

Variable recombination rates

class msprime.**RecombinationMap**(*positions*, *rates*, *num_loci=None*)

A RecombinationMap represents the changing rates of recombination along a chromosome. This is defined via two lists of numbers: `positions` and `rates`, which must be of the same length. Given an index j in these lists, the rate of recombination per base per generation is `rates[j]` over the interval `positions[j]` to `positions[j + 1]`. Consequently, the first position must be zero, and by convention the last rate value is also required to be zero (although it does not used).

Parameters

- **positions** (*list*) – The positions (in bases) denoting the distinct intervals where recombination rates change. These can be floating point values.
- **rates** (*list*) – The list of rates corresponding to the supplied `positions`. Recombination rates are specified per base, per generation.
- **num_loci** (*int*) – The maximum number of non-recombining loci in the underlying simulation. By default this is set to the largest possible value, allowing the maximum resolution in the recombination process. However, for a finite sites model this can be set to smaller values.

classmethod read_hapmap(*filename*)

Parses the specified file in HapMap format. These files must contain a single header line (which is ignored), and then each subsequent line denotes a position/rate pair. Positions are in units of bases, and recombination rates in centimorgans/Megabase. The first column in this file is ignored, as are subsequence columns after the Position and Rate. A sample of this format is as follows:

```
Chromosome  Position(bp)    Rate(cM/Mb)     Map(cM)
chr1        55550   2.981822        0.000000
chr1        82571   2.082414        0.080572
chr1        88169   2.081358        0.092229
chr1        254996  3.354927        0.439456
chr1        564598  2.887498        1.478148
```

Parameters **filename** (*str*) – The name of the file to be parsed. This may be in plain text or gzipped plain text.

1.3.3 Processing results

The *TreeSequence* class represents a sequence of correlated trees output by a simulation. The *SparseTree* class represents a single tree in this sequence.

msprime.NULL_NODE = -1
Special reserved value, representing the null node. If the parent of a given node is null, then this node is a root. Similarly, if the children of a node are null, this node is a leaf.

msprime.NULL_POPULATION = -1
Special reserved value, representing the null population ID. If the population associated with a particular tree node is not defined, or population information was not available in the underlying tree sequence, then this value will be returned by *SparseTree.get_population()*.

msprime.FORWARD = 1
Constant representing the forward direction of travel (i.e., increasing coordinate values).

msprime.REVERSE = -1
Constant representing the reverse direction of travel (i.e., decreasing coordinate values).

msprime.**load**(*path*)
Loads a tree sequence from the specified file path. This file must be in the HDF5 file format produced by the *TreeSequence.dump()* method.

Parameters **path** (*str*) – The file path of the HDF5 file containing the tree sequence we wish to load.

Returns The tree sequence object containing the information stored in the specified file path.

Return type *msprime.TreeSequence*

msprime.**load_txt**(*records_file*, *mutations_file=None*)
Loads a tree sequence from the specified file paths. The files input here are in a simple whitespace delimited tabular format such as output by the *TreeSequence.write_records()* and *TreeSequence.write_mutations()* methods. This method is intended as a convenient interface for importing external data into msprime; the HDF5 based file format using by *msprime.load()* will be many times more efficient that using the text based formats.

The `records_file` must be a text file with six whitespace delimited columns. Each line in the file must contain at least this many columns, and each line will be stored as a single coalescence record. The columns correspond to the `left`, `right`, `node`, `children`, `time` and `population` fields as described in the *TreeSequence.records()* method. The `left`, `right` and `time` fields are parsed as base 10 floating point values, and the `node` and `population` fields are parsed as base 10 integers. The `children` field is a comma-separated list of base 10 integer values, and must contain at least two elements. The file may optionally begin with a header line; if the first line begins with the text "left" it will be ignored.

Records must be listed in the file in non-decreasing order of the time field. Within a record, children must be listed in increasing order of node value. The left and right coordinates must be non-negative values.

An example of a simple tree sequence for four samples with three distinct trees is:

```
left    right   node    children    time    population
2       10      4       2,3         0.071    0
0       2       5       1,3         0.090    0
2       10      5       1,4         0.090    0
0       7       6       0,5         0.170    0
7       10      7       0,5         0.202    0
0       2       8       2,6         0.253    0
```

This example is equivalent to the tree sequence illustrated in Figure 4 of the PLoS Computational Biology paper. Nodes are given here in time order (since this is a backwards-in-time tree sequence), but they may be allocated in any order. In particular, left-to-right tree sequences are fully supported. However, the smallest value in the `node` column must be equal to the sample size, and there must not be 'gaps' in the node address space.

The optional `mutations_file` has a similiar format, but contains only two columns. These correspond to the `position` and `node` fields as described in the *TreeSequence.mutations()* method. The `position`

field is parsed as a base 10 floating point value, and the `node` field is parsed as a base 10 integer. The file may optionally begin with a header line; if the first line begins with the text "position" it will be ignored.

Mutations must be listed in non-decreasing order of position, and the nodes must refer to a node defined by the records. Mutations defined over the root or a node not present in a local tree will lead to an error being produced during tree traversal (e.g. in the *TreeSequence.trees()* method, but also in many other methods).

An example of a mutations file for the tree sequence defined in the previous example is:

```
position    node
0.1         0
8.5         4
```

Parameters

- **records_file** (*str*) – The path of the text file containing the coalescence records for the desired tree sequence.
- **mutations_file** (*str*) – The path of the text file containing the mutation records for the desired tree sequence. This argument is optional and defaults to None.

Returns The tree sequence object containing the information stored in the specified file paths.

Return type *msprime.TreeSequence*

class msprime.**TreeSequence**

A TreeSequence represents the information generated in a coalescent simulation. This includes all the trees across the simulated region, along with the mutations (if any are present).

breakpoints()

Returns an iterator over the breakpoints along the chromosome, including the two extreme points 0 and L. This is equivalent to

```
>>> [0] + [t.get_interval()[1] for t in self.trees()]
```

although we do not build an explicit list.

Returns An iterator over all the breakpoints along the simulated sequence.

Return type iter

diffs()

Returns an iterator over the differences between adjacent trees in this tree sequence. Each diff returned by this method is a tuple of the form *(length, records_out, records_in)*. The *length* is the length of the genomic interval covered by the current tree, and is equivalent to the value returned by *msprime.SparseTree.get_length()*. The *records_out* value is list of (u, c, t) tuples, and corresponds to the coalescence records that have been invalidated by moving to the current tree. As in the *records()* method, u is a tree node, c is a tuple containing its children, and t is the time the event occurred. These records are returned in time-decreasing order, such that the record affecting the highest parts of the tree (i.e., closest to the root) are returned first. The *records_in* value is also a list of (u, c, t) tuples, and these describe the records that must be applied to create the tree covering the current interval. These records are returned in time-increasing order, such that the records affecting the lowest parts of the tree (i.e., closest to the leaves) are returned first.

Returns An iterator over the diffs between adjacent trees in this tree sequence.

Return type iter

dump(*path*, *zlib_compression=False*)

Writes the tree sequence to the specified file path.

Parameters

- **path** (*str*) – The file path to write the TreeSequence to.
- **zlib_compression** (*bool*) – If True, use HDF5's native compression when storing the data leading to smaller file size. When loading, data will be decompressed transparently, but load times will be significantly slower.

get_num_mutations()
Returns the number of mutations in this tree sequence. See the *msprime.TreeSequence.mutations()* method for details on how mutations are defined.

Returns The number of mutations in this tree sequence.

Return type int

get_num_nodes()
Returns the number of nodes in this tree sequence. This 1 + the largest value u such that u is a node in any of the constituent trees.

Returns The total number of nodes in this tree sequence.

Return type int

get_num_records()
Returns the number of coalescence records in this tree sequence. See the *records()* method for details on these objects.

Returns The number of coalescence records defining this tree sequence.

Return type int

get_num_trees()
Returns the number of distinct trees in this tree sequence. This is equal to the number of trees returned by the *trees()* method.

Returns The number of trees in this tree sequence.

Return type int

get_pairwise_diversity(*samples=None*)
Returns the value of pi, the pairwise nucleotide site diversity. If *samples* is specified, calculate the diversity within this set.

Parameters **samples** (*iterable*) – The set of samples within which we calculate the diversity. If None, calculate diversity within the entire sample.

Returns The pairwise nucleotide site diversity.

Return type float

get_population(*sample*)
Returns the population ID for the specified sample ID.

Parameters **sample** (*int*) – The sample ID of interest.

Returns The population ID where the specified sample was drawn. Returns NULL_POPULATION if no population information is available.

Return type int

get_sample_size()
Returns the sample size for this tree sequence. This is the number of leaf nodes in each tree.

Returns The number of leaf nodes in the tree sequence.

Return type int

get_samples(*population_id=None*)
Returns the samples matching the specified population ID.

Parameters **population_id** (*int*) – The population of interest. If None, return all samples.

Returns The ID of the population we wish to find samples from. If None, return samples from all populations.

Return type list

get_sequence_length()
Returns the sequence length in this tree sequence. This defines the genomic scale over which tree coordinates are defined. Given a tree sequence with a sequence length L, the constituent trees will be defined over the half-closed interval $(0, L]$. Each tree then covers some subset of this interval — see *msprime.SparseTree.get_interval()* for details.

Returns The length of the sequence in this tree sequence in bases.

Return type float

get_time(*sample*)
Returns the time that the specified sample ID was sampled at.

Parameters **sample** (*int*) – The sample ID of interest.

Returns The time at which the specified sample was drawn.

Return type int

haplotypes()
Returns an iterator over the haplotypes resulting from the trees and mutations in this tree sequence as a string of '1's and '0's. The iterator returns a total of n strings, each of which contains s characters (n is the sample size returned by *msprime.TreeSequence.get_sample_size()* and s is the number of mutations returned by *msprime.TreeSequence.get_num_mutations()*). The first string returned is the haplotype for sample *0*, and so on.

Returns An iterator over the haplotype strings for the samples in this tree sequence.

Return type iter

mutations()
Returns an iterator over the mutations in this tree sequence. Each mutation is represented as a tuple (x, u, j) where x is the position of the mutation in the sequence in chromosome coordinates, u is the node over which the mutation occurred and j is the zero-based index of the mutation within the overall tree sequence. Mutations are returned in non-decreasing order of position and increasing index.

Each mutation returned is an instance of `collections.namedtuple()`, and may be accessed via the attributes `position`, `node` and `index` as well as the usual positional approach. This is the recommended interface for working with mutations as it is both more readable and also ensures that code is forward compatible with future extensions.

Returns An iterator of all (x, u, j) tuples defining the mutations in this tree sequence.

Return type iter

records()
Returns an iterator over the coalescence records in this tree sequence in time-sorted order. Each record is a tuple (l, r, u, c, t, d) defining the assignment of a tree node across an interval. The range of this record is the half-open genomic interval $[l, r)$, such that it applies to all positions $l \leq x < r$. Each record represents the assignment of a pair of children c to a parent parent u. This assignment happens at t generations in the past within the population with ID d. If population information was not stored for this tree sequence then the population ID will be `NULL_POPULATION`.

Each record returned is an instance of `collections.namedtuple()`, and may be accessed via the attributes `left`, `right`, `node`, `children`, `time` and `population`, as well as the usual positional approach. For example, if we wished to print out the genomic length of each record, we could write:

```
>>> for record in tree_sequence.records():
>>>     print(record.right - record.left)
```

Returns An iterator of all (l, r, u, c, t, d) tuples defining the coalescence records in this tree sequence.

Return type iter

set_mutations(*mutations*)

Sets the mutations in this tree sequence to the specified list of mutations. Each entry in the list must be either a `Mutation` named-tuple instance (as returned by the *TreeSequence.mutations()* method) or tuple of the form $(x, u, ...)$, where x is a floating point value defining a genomic position and u is an integer defining a tree node. A genomic position x must satisfy $0 \leq x < L$ where L is the sequence length (see *get_sequence_length()*). A node u must satisfy $0 < u < N$ where N is the number of nodes in the tree sequence (see *get_num_nodes()*). Values other than `position` and `node` in the input tuples are ignored.

Parameters **mutations** (*list*) – The list of mutations to be assigned to this tree sequence.

trees(*tracked_leaves=None*, *leaf_counts=True*, *leaf_lists=False*)

Returns an iterator over the trees in this tree sequence. Each value returned in this iterator is an instance of *SparseTree*.

The `leaf_counts` and `leaf_lists` parameters control the features that are enabled for the resulting trees. If `leaf_counts` is True, then it is possible to count the number of leaves underneath a particular node in constant time using the *get_num_leaves()* method. If `leaf_lists` is True a more efficient algorithm is used in the *SparseTree.leaves()* method.

The `tracked_leaves` parameter can be used to efficiently count the number of leaves in a given set that exist in a particular subtree using the *SparseTree.get_num_tracked_leaves()* method. It is an error to use the `tracked_leaves` parameter when the `leaf_counts` flag is False.

Warning Do not store the results of this iterator in a list! For performance reasons, the same underlying object is used for every tree returned which will most likely lead to unexpected behaviour.

Parameters

- **tracked_leaves** (*list*) – The list of leaves to be tracked and counted using the *SparseTree.get_num_tracked_leaves()* method.
- **leaf_counts** (*bool*) – If True, support constant time leaf counts via the *SparseTree.get_num_leaves()* and *SparseTree.get_num_tracked_leaves()* methods.
- **leaf_lists** (*bool*) – If True, provide more efficient access to the leaves beneath a give node using the *SparseTree.leaves()* method.

Returns An iterator over the sparse trees in this tree sequence.

Return type iter

variants(*as_bytes=False*)

Returns an iterator over the variants in this tree sequence. Each variant corresponds to a single mutation and is represented as a tuple (x, u, j, g). The values of x, u and j are identical to the values returned by the *TreeSequence.mutations()* method, and g represents the sample genotypes for this variant. Thus,

$g[k]$ is the observed state for sample k at this site; zero represents the ancestral type and one the derived type.

Each variant returned is an instance of `collections.namedtuple()`, and may be accessed via the attributes `position`, `node`, `index` and `genotypes` as well as the usual positional approach. This is the recommended interface for working with variants as it is both more readable and also ensures that code is forward compatible with future extensions.

The returned genotypes may be either a numpy array of 1 byte unsigned integer 0/1 values, or a Python bytes object of '0'/'1' ASCII characters. This behaviour is controller by the `as_bytes` parameter. The default behaviour is to return a numpy array, which is substantially more efficient.

Warning The same numpy array is used to represent genotypes between iterations, so if you wish the store the results of this iterator you **must** take a copy of the array. This warning does not apply when `as_bytes` is True, as a new bytes object is allocated for each variant.

Parameters **as_bytes** (*bool*) – If True, the genotype values will be returned as a Python bytes object. This is useful in certain situations (i.e., directly printing the genotypes) or when numpy is not available. Otherwise, genotypes are returned as a numpy array (the default).

Returns An iterator of all (x, u, j, g) tuples defining the variants in this tree sequence.

write_mutations(*output*, *header=True*, *precision=6*)

Writes the mutations for this tree sequence to the specified file in a tab-separated format. If `header` is True, the first line of this file contains the names of the columns, i.e., `position` and `node`. The `position` field describes the location of the mutation along the sequence in chromosome coordinates, and the `node` field defines the node over which the mutation occurs. After the optional header, the records are written to the file in tab-separated form in order of non-decreasing position. The `position` field is a base 10 floating point value printed to the specified `precision`. The `node` field is a base 10 integer.

Example usage:

```
>>> with open("mutations.txt", "w") as mutations_file:
>>>     tree_sequence.write_mutations(mutations_file)
```

Parameters

- **output** (*File*) – The file-like object to write the tab separated output.
- **header** (*bool*) – If True, write a header describing the column names in the output.
- **precision** (*int*) – The number of decimal places to print out for floating point columns.

write_records(*output*, *header=True*, *precision=6*)

Writes the records for this tree sequence to the specified file in a tab-separated format. If `header` is True, the first line of this file contains the names of the columns, i.e., `left`, `right`, `node`, `children`, `time` and `population`. After the optional header, the records are written to the file in tab-separated form in order of non-decreasing time. The `left`, `right` and `time` fields are base 10 floating point values printed to the specified `precision`. The `node` and `population` fields are base 10 integers. The `children` column is a comma-separated list of base 10 integers, which must contain at least two values.

Example usage:

```
>>> with open("records.txt", "w") as records_file:
>>>     tree_sequence.write_records(records_file)
```

Parameters

- **output** (*File*) – The file-like object to write the tab separated output.

- **header** (*bool*) – If True, write a header describing the column names in the output.
- **precision** (*int*) – The number of decimal places to print out for floating point columns.

write_vcf(*output*, *ploidy=1*)

Writes a VCF formatted file to the specified file-like object. If a ploidy value is supplied, allele values are combined among adjacent samples to form a phased genotype of the required ploidy. For example, if we have a ploidy of 2 and a sample of size 6, then we will have 3 diploid samples in the output, consisting of the combined alleles for samples [0, 1], [2, 3] and [4, 5]. If we had alleles 011110 at a particular variant, then we would output the genotypes 0|1, 1|1 and 1|0 in VCF. Sample names are generated by appending the index to the prefix `msp_` such that we would have the sample names `msp_0`, `msp_1` and `msp_2` in the running example.

Example usage:

```
>>> with open("output.vcf", "w") as vcf_file:
>>>     tree_sequence.write_vcf(vcf_file, 2)
```

Parameters

- **output** (*File*) – The file-like object to write the VCF output.
- **ploidy** (*int*) – The ploidy of the individual samples in the VCF. This sample size must be divisible by ploidy.

class msprime.**SparseTree**

A SparseTree is a single tree in a *TreeSequence*. In a sparse tree for a sample of size n, the leaves are nodes 0 to $n - 1$ inclusive and internal nodes are integers $\geq n$. The value of these nodes is strictly increasing as we ascend the tree and the root of the tree is the node with the largest value that is reachable from the leaves. Each node in the tree has a parent which is obtained using the *get_parent()* method. The parent of the root node is the `NULL_NODE`, -1. Similarly, each internal node has a pair of children, which are obtained using the *get_children()* method. Each node in the tree has a time associated with it in generations. This value is obtained using the *SparseTree.get_time()* method.

Sparse trees are not intended to be instantiated directly, and are obtained as part of a *TreeSequence* using the *trees()* method.

draw(*path*, *width=200*, *height=200*, *show_times=False*)

Draws a representation of this tree to the specified path in SVG format.

Parameters

- **path** (*str*) – The path to the file to write the SVG.
- **width** (*int*) – The width of the image in pixels.
- **height** (*int*) – The height of the image in pixels.
- **show_times** (*bool*) – If True, show time labels at each internal node.

get_branch_length(*u*)

Returns the length of the branch (in generations) joining the specified node to its parent. This is equivalent to

```
>>> tree.get_time(tree.get_parent(u)) - tree.get_time(u)
```

Note that this is not related to the value returned by *get_length()*, which describes the length of the interval covered by the tree in genomic coordinates.

Parameters **u** (*int*) – The node of interest.

Returns The branch length from u to its parent.

Return type float

get_children(*u*)
Returns the children of the specified node as a tuple (v, w). For internal nodes, this tuple is always in sorted order such that $v < w$. If u is a leaf or is not a node in the current tree, return the tuple (NULL_NODE, NULL_NODE).

Parameters **u** (*int*) – The node of interest.

Returns The children of u as a pair of integers

Return type tuple

get_index()
Returns the index this tree occupies in the parent tree sequence. This index is zero based, so the first tree in the sequence has index 0.

Returns The index of this tree.

Return type int

get_interval()
Returns the coordinates of the genomic interval that this tree represents the history of. The interval is returned as a tuple (l, r) and is a half-open interval such that the left coordinate is inclusive and the right coordinate is exclusive. This tree therefore applies to all genomic locations x such that $l \leq x < r$.

Returns A tuple (l, r) representing the left-most (inclusive) and right-most (exclusive) coordinates of the genomic region covered by this tree.

Return type tuple

get_length()
Returns the length of the genomic interval that this tree represents. This is defined as $r - l$, where (l, r) is the genomic interval returned by *get_interval()*.

Returns The length of the genomic interval covered by this tree.

Return type int

get_mrca(*u*, *v*)
Returns the most recent common ancestor of the specified nodes.

Parameters

- **u** (*int*) – The first node.
- **v** (*int*) – The second node.

Returns The most recent common ancestor of u and v.

Return type int

get_num_leaves(*u*)
Returns the number of leaves in this tree underneath the specified node.

If the *TreeSequence.trees()* method is called with leaf_counts=True this method is a constant time operation. If not, a slower traversal based algorithm is used to count the leaves.

Parameters **u** (*int*) – The node of interest.

Returns The number of leaves in the subtree rooted at u.

Return type int

get_num_mutations()
Returns the number of mutations on this tree.

Returns The number of mutations on this tree.

Return type int

get_num_tracked_leaves(*u*)
Returns the number of leaves in the set specified in the `tracked_leaves` parameter of the *TreeSequence.trees()* method underneath the specified node. This is a constant time operation.

Parameters **u** (*int*) – The node of interest.

Returns The number of leaves within the set of tracked leaves in the subtree rooted at u.

Return type int

Raises **RuntimeError** – if the *TreeSequence.trees()* method is not called with `leaf_counts=True`.

get_parent(*u*)
Returns the parent of the specified node. Returns the `NULL_NODE` -1 if u is the root or is not a node in the current tree.

Parameters **u** (*int*) – The node of interest.

Returns The parent of u.

Return type int

get_population(*u*)
Returns the population associated with the specified node. For leaf nodes this is the population of the sample, and for internal nodes this is the population where the corresponding coalescence occured. If the specified node is not a member of this tree or population level information was not stored in the tree sequence, `NULL_POPULATION` is returned.

Parameters **u** (*int*) – The node of interest.

Returns The ID of the population associated with node u.

Return type int

get_root()
Returns the root of this tree.

Returns The root node.

Return type int

get_sample_size()
Returns the sample size for this tree. This is the number of leaf nodes in the tree.

Returns The number of leaf nodes in the tree.

Return type int

get_time(*u*)
Returns the time of the specified node in generations. Returns 0 if u is a leaf or is not a node in the current tree.

Parameters **u** (*int*) – The node of interest.

Returns The time of u.

Return type float

get_tmrca(*u*, *v*)
Returns the time of the most recent common ancestor of the specified nodes. This is equivalent to:

```
tree.get_time(tree.get_mrca(u, v))
```

Parameters

- **u** (*int*) – The first node.
- **v** (*int*) – The second node.

Returns The time of the most recent common ancestor of u and v.

Return type float

get_total_branch_length()
Returns the sum of all the branch lengths in this tree (in units of generations). This is equivalent to

```
>>> sum(
>>>     tree.get_branch_length(u) for u in tree.nodes()
>>>     if u != tree.get_root())
```

Returns The sum of all the branch lengths in this tree.

is_internal(*u*)
Returns True if the specified node is not a leaf.

Parameters **u** (*int*) – The node of interest.

Returns True if u is not a leaf node.

Return type bool

is_leaf(*u*)
Returns True if the specified node is a leaf. A node u is a leaf if $0 \leq u < n$ for a sample size n.

Parameters **u** (*int*) – The node of interest.

Returns True if u is a leaf node.

Return type bool

leaves(*u*)
Returns an iterator over all the leaves in this tree underneath the specified node.

If the *TreeSequence.trees()* method is called with `leaf_lists=True`, this method uses an efficient algorithm to find the leaves. If not, a simple traversal based method is used.

Parameters **u** (*int*) – The node of interest.

Returns An iterator over all leaves in the subtree rooted at u.

Return type iterator

mutations()
Returns an iterator over the mutations in this tree. Each mutation is represented as a tuple (x, u, j) where x is the position of the mutation in the sequence in chromosome coordinates, u is the node over which the mutation occurred and j is the zero-based index of the mutation within the overall tree sequence. Mutations are returned in non-decreasing order of position and increasing index.

Each mutation returned is an instance of `collections.namedtuple()`, and may be accessed via the attributes `position`, `node` and `index` as well as the usual positional approach. This is the recommended interface for working with mutations as it is both more readable and also ensures that code is forward compatible with future extensions.

Returns An iterator of all (x, u, j) tuples defining the mutations in this tree.

Return type iter

nodes (*root=None*, *order='preorder'*)

Returns an iterator over the nodes in this tree. If the root parameter is provided, iterate over the nodes in the subtree rooted at this node. If this is None, iterate over all nodes. If the order parameter is provided, iterate over the nodes in required tree traversal order.

Parameters

- **root** (*int*) – The root of the subtree we are traversing.
- **order** (*str*) – The traversal ordering. Currently only 'preorder' is supported.

Return type iterator

1.3.4 Calculating statistics

The `msprime` API provides methods for efficiently calculating population genetics statistics from a given *TreeSequence*.

class msprime.**LdCalculator** (*tree_sequence*)

Class for calculating linkage disequilibrium coefficients between pairs of mutations in a *TreeSequence*. This class requires the numpy library.

This class supports multithreaded access using the Python `threading` module. Separate instances of *LdCalculator* referencing the same tree sequence can operate in parallel in multiple threads. See the *Working with threads* section in the *Tutorial* for an example of how use multiple threads to calculate LD values efficiently.

Parameters **tree_sequence** (TreeSequence) – The tree sequence containing the mutations we are interested in.

get_r2 (*a*, *b*)

Returns the value of the r^2 statistic between the pair of mutations at the specified indexes. This method is *not* an efficient method for computing large numbers of pairwise; please use either *get_r2_array()* or *get_r2_matrix()* for this purpose.

Parameters

- **a** (*int*) – The index of the first mutation.
- **b** (*int*) – The index of the second mutation.

Returns The value of r^2 between the mutations at indexes `a` and `b`.

Return type float

get_r2_array (*a*, *direction=1*, *max_mutations=None*, *max_distance=None*)

Returns the value of the r^2 statistic between the focal mutation at index a and a set of other mutations. The method operates by starting at the focal mutation and iterating over adjacent mutations (in either the forward or backwards direction) until either a maximum number of other mutations have been considered (using the `max_mutations` parameter), a maximum distance in sequence coordinates has been reached (using the `max_distance` parameter) or the start/end of the sequence has been reached. For every mutation b considered, we then insert the value of r^2 between a and b at the corresponding index in an array, and return the entire array. If the returned array is x and `direction` is `msprime.FORWARD`

then $x[0]$ is the value of the statistic for a and $a + 1$, $x[1]$ the value for a and $a + 2$, etc. Similarly, if `direction` is `msprime.REVERSE` then $x[0]$ is the value of the statistic for a and $a - 1$, $x[1]$ the value for a and $a - 2$, etc.

Parameters

- **a** (*int*) – The index of the focal mutation.
- **direction** (*int*) – The direction in which to travel when examining other mutations. Must be either `msprime.FORWARD` or `msprime.REVERSE`. Defaults to `msprime.FORWARD`.
- **max_mutations** (*int*) – The maximum number of mutations to return r^2 values for. Defaults to as many mutations as possible.
- **max_distance** (*float*) – The maximum absolute distance between the focal mutation and those for which r^2 values are returned.

Returns An array of double precision floating point values representing the r^2 values for mutations in the specified direction.

Return type numpy.ndarray

Warning For efficiency reasons, the underlying memory used to store the returned array is shared between calls. Therefore, if you wish to store the results of a single call to `get_r2_array()` for later processing you **must** take a copy of the array!

get_r2_matrix()

Returns the complete $m \times m$ matrix of pairwise r^2 values in a tree sequence with m mutations.

Returns An 2 dimensional square array of double precision floating point values representing the r^2 values for all pairs of mutations.

Return type numpy.ndarray

1.4 Command line interface

Two command-line applications are provided with `msprime`: *msp* and *mspms*. The **msp** program is an experimental interface for interacting with the library, and is a POSIX compliant command line interface. The **mspms** program is a fully-**ms** compatible interface. This is useful for those who wish to get started quickly with using the library, and also as a means of plugging `msprime` into existing work flows. However, there is a substantial overhead involved in translating data from `msprime`'s native history file into legacy formats, and so new code should use the *Python API* where possible.

1.4.1 msp

The `msp` program provides a convenient interface to the *msprime API*. It is based on subcommands that either generate or consume a *history file*. The `simulate` subcommand runs a simulation storing the results in a file. The other commands are concerned with converting this file into other formats.

Warning: This tool is very new, and the interface may need to change over time. This should be considered an alpha feature!

msp simulate

msp simulate provides a command line interface to the *msprime.simulate()* API function. Using the parameters provided at the command line, we run a simulation and then save the resulting tree sequence to the file provided as an argument.

```
usage: msp simulate [-h] [--length LENGTH]
                    [--recombination-rate RECOMBINATION_RATE]
                    [--mutation-rate MUTATION_RATE]
                    [--effective-population-size EFFECTIVE_POPULATION_SIZE]
                    [--random-seed RANDOM_SEED] [--compress]
                    sample_size history_file
```

Positional arguments:

sample_size	The number of individuals in the sample
history_file	The msprime history file in HDF5 format

Options:

--length, -L	The length of the simulated region in base pairs.
--recombination-rate, -r	The recombination rate per base per generation
--mutation-rate, -u	The mutation rate per base per generation
--effective-population-size, -N	The effective population size Ne
--random-seed, -s	The random seed. If not specified one is chosen randomly
--compress, -z	Enable HDF5's transparent zlib compression

Note: The way in which recombination and mutation rates are specified is different to **ms**. In **ms** these rates are scaled by the length of the simulated region, whereas we use rates per unit distance. The rationale for this change is simplify running simulations on a variety of sequence lengths, so that we need to change only parameter and not three simultaneously.

---

msp upgrade

msp upgrade is a command line tool to convert tree sequence files written by older versions of msprime to the latest version. This tool requires h5py, so please ensure that it is installed. The upgrade process involves creating a new tree sequence file from the records stored in the older file and is non-destructive.

```
usage: msp upgrade [-h] source destination
```

Positional arguments:

source	The source msprime history file in legacy HDF5 format
destination	The filename of the upgraded copy.

msp records

msp records is a command line interface to the *msprime.TreeSequence.write_records()* method. It prints out the variants stored in the history file in VCF format.

```
usage: msp records [-h] [--header] [--precision PRECISION] history_file
```

Positional arguments:

history_file	The msprime history file in HDF5 format

Options:

--header, -H	Print a header line in the output.
--precision, -p	The number of decimal places to print in records

msp vcf

msp vcf is a command line interface to the *msprime.TreeSequence.write_vcf()* method. It prints out the coalescence vcf in a history file in a tab-delimited text format.

```
usage: msp vcf [-h] [--ploidy PLOIDY] history_file
```

Positional arguments:

history_file	The msprime history file in HDF5 format

Options:

--ploidy, -P	The ploidy level of samples

msp mutations

msp mutations is a command line interface to the *msprime.TreeSequence.mutations()* method. It prints out the coalescence mutations in a history file in a tab-delimited text format.

```
usage: msp mutations [-h] [--header] [--precision PRECISION] history_file
```

Positional arguments:

history_file	The msprime history file in HDF5 format

Options:

--header, -H	Print a header line in the output.
--precision, -p	The number of decimal places to print in records

msp newick

msp mutations prints out the marginal genealogies in the tree sequence in newick format.

```
usage: msp mutations [-h] [--header] [--precision PRECISION] history_file
```

Positional arguments:

history_file	The msprime history file in HDF5 format

Options:

--header, -H	Print a header line in the output.
--precision, -p	The number of decimal places to print in records

1.4.2 mspms

The **mspms** program is an **ms**-compatible command line interface to the `msprime` library. This interface should be useful for legacy applications, where it can be used as a drop-in replacement for **ms**. This interface is not recommended for new applications, particularly if the simulated trees are required as part of the output as Newick is very inefficient. The *Python API* is the recommended interface, providing direct access to the structures used within `msprime`.

Supported Features

mspms supports a subset of **ms**'s functionality. Please open an issue on GitHub if there is a feature of **ms** that you would like to see added. We currently support:

- Basic functionality (sample size, replicates, tree and haplotype output);
- Recombination (via the `-r` option);
- Spatial structure with arbitrary migration matrices;
- Support for **ms** demographic events. (The implementation of the `-es` option is limited, and has restrictions on how it may be combined with other options.)

Gene-conversion is not currently supported, but is planned for a future release.

Argument details

This section provides the detailed listing of the arguments to **mspms** (also available via `mspms --help`). See the documentation for ms for details on how these values should be interpreted.

mspms is an ms-compatible interface to the msprime library. It simulates the coalescent with recombination for a variety of demographic models and outputs the results in a text-based format. It supports a subset of the functionality available in ms and aims for full compatibility.

```
usage: mspms [-h] [-V] [--mutation-rate theta] [--trees]
             [--recombination rho num_loci] [--structure value [value ...]]
             [--migration-matrix-entry dest source rate]
             [--migration-matrix entry [entry ...]]
             [--migration-rate-change t x]
             [--migration-matrix-entry-change time dest source rate]
             [--migration-matrix-change entry [entry ...]]
             [--growth-rate alpha]
             [--population-growth-rate population_id alpha]
             [--population-size population_id size]
             [--growth-rate-change t alpha]
             [--population-growth-rate-change t population_id alpha]
             [--size-change t x] [--population-size-change t population_id x]
             [--population-split t dest source]
             [--admixture t population_id proportion]
             [--random-seeds x1 x2 x3] [--precision PRECISION]
             sample_size num_replicates
```

Positional arguments:

sample_size	The number of individuals in the sample
num_replicates	Number of independent replicates

Options:

-V, --version	show program's version number and exit

--mutation-rate, -t Mutation rate theta=4*N0*mu

--trees, -T Print out trees in Newick format

--recombination, -r Recombination at rate rho=4*N0*r where r is the rate of recombination between the ends of the region being simulated; num_loci is the number of sites between which recombination can occur

--structure, -I Sample from populations with the specified deme structure. The arguments are of the form 'num_populations n1 n2 ... [4N0m]', specifying the number of populations, the sample configuration, and optionally, the migration rate for a symmetric island model

--migration-matrix-entry, -m Sets an entry M[dest, source] in the migration matrix to the specified rate. source and dest are (1-indexed) population IDs. Multiple options can be specified.

--migration-matrix, -ma Sets the migration matrix to the specified value. The entries are in the order M[1,1], M[1, 2], ..., M[2, 1],M[2, 2], ..., M[N, N], where N is the number of populations.

--migration-rate-change, -eM Set the symmetric island model migration rate to x / (npop - 1) at time t

--migration-matrix-entry-change, -em Sets an entry M[dest, source] in the migration matrix to the specified rate at the specified time. source and dest are (1-indexed) population IDs.

--migration-matrix-change, -ema Sets the migration matrix to the specified value at time t.The entries are in the order M[1,1], M[1, 2], ..., M[2, 1],M[2, 2], ..., M[N, N], where N is the number of populations.

--growth-rate, -G Set the growth rate to alpha for all populations.

--population-growth-rate, -g Set the growth rate to alpha for a specific population.

--population-size, -n Set the size of a specific population to size*N0.

--growth-rate-change, -eG Set the growth rate for all populations to alpha at time t

--population-growth-rate-change, -eg Set the growth rate for a specific population to alpha at time t

--size-change, -eN Set the population size for all populations to x * N0 at time t

--population-size-change, -en Set the population size for a specific population to x * N0 at time t

--population-split, -ej Move all lineages in population dest to source at time t. Forwards in time, this corresponds to a population split in which lineages in source split into dest. All migration rates for population source are set to zero.

--admixture, -es Split the specified population into a new population, such that the specified proportion of lineages remains in the population population_id. Forwards in time this corresponds to an admixture event. The new population has ID num_populations + 1. Migration rates to and from the new population are set to 0, and growth rate is 0 and the population size for the new population is N0.

--random-seeds, -seeds Random seeds (must be three integers)

--precision, -p Number of values after decimal place to print

If you use msprime in your work, please cite the following paper: Jerome Kelleher, Alison M Etheridge and Gilean McVean (2016), "Efficient Coalescent Simulation and Genealogical Analysis for Large Sample Sizes", PLoS Comput Biol 12(5): e1004842. doi: 10.1371/journal.pcbi.1004842

1.5 Tree Sequence File Format

The correlated trees output by a coalescent simulation are stored very concisely in `msprime` as a sequence of coalescent records. To make this information as efficient and easy as possible to use, we store the data in a HDF5 based file format. This page fully documents this format allowing efficient and convenient access to the genealogical data generated by `msprime` outside of the native *Python API*. Using the specification defined here, it should be straightforward to access tree sequence information in any language with HDF5 support.

1.5.1 Structure

The file format is broken into a number of groups. Each group contains datasets to define the data along with attributes to provide necessary contextual information.

The root group contains one attributes, `format_version`. This is a pair `(major, minor)` describing the file format version. This document describes version 3.1.

Path	Type	Dim	Description
/format_version	H5T_STD_U32LE	2	The (major, minor) file format version.

Provenance dataset

The provenance dataset records information relating the the provenance of a particular tree sequence file. When a tree sequence file is generated all the information required to reproduce the file should be encoded as a string and stored in this dataset. Subsequent modifications to the file should be also be recorded and appended to the list of strings.

The format of these strings is implementation defined. In the current version of `msprime` provenance information is encoded as JSON. This information is incomplete, and will be updated in future versions.

Path	Type	Dim	Description
/provenance	H5T_STRING	Scalar	Provenance information.

Mutations group

The `mutations` group is optional, and describes the location of mutations with respect to tree nodes and their positions along the sequence. Each mutation consists of a node (which must be defined in the `trees` group) and a position. Positions are defined as a floating point value to allow us to express infinite sites mutations. A mutation position x is defined on the same scale as the genomic coordinates for trees, and so we must have $0 \leq x < L$, where L is the largest value in the `/trees/breakpoints` dataset.

As for the coalescence records in the `trees` group, mutation records are stored as seperate vectors for efficiency reasons. Mutations must be stored in nondecreasing order of position.

Path	Type	Dim
/mutations/node	H5T_STD_U32LE	M
/mutations/position	H5T_IEEE_F64LE	M

Trees group

The `trees` group is mandatory and describes the topology of the tree sequence. The `trees` group contains a number of nested groups and datasets, which we will describe in turn.

Breakpoints dataset

The `/trees/breakpoints` dataset records the floating point positions of the breakpoints between trees in the tree sequence, and the flanking positions 0 and L. Positions in the `/trees/records` group refer to (zero based) indexes into this array. The first breakpoint must be zero, and they must be listed in increasing order.

Path	Type
/trees/breakpoints	H5T_IEEE_F64LE

Nodes group

The `/trees/nodes` group records information about the individual nodes in a tree sequence. Leaf nodes (from 0 to $n-1$) represent the samples and internal nodes ($\geq n$) represent their ancestors. Each node corresponds to a particular individual that lived at some time time in the history of the sample. The `nodes` group is used to record information about these individuals.

Path	Type
/trees/nodes/population	H5T_STD_U8LE
/trees/nodes/time	H5T_IEEE_F64LE

Records group

The `/trees/records` group stores the individual coalesence records. Each record consists of four pieces of information: the left and right coordinates of the coalescing interval, the list of child nodes and the parent node.

The `left` and `right` datasets are indexes into the `/trees/breakpoints` dataset and define the genomic interval over which the record applies. The interval is half-open, so that the left coordinate is inclusive and the right coordinate is exclusive.

The `node` dataset records the parent node of the record, and is an index into the `/trees/nodes` group.

The `num_children` dataset records the number of children for a particular record. The `children` dataset then records the actual child nodes for each coalescence record. This 1-dimensional array lists the child nodes for every record in order, and therefore by using the `num_children` array we can efficiently recover the actual children involved in each event. Within a given event, child nodes must be sorted in increasing order. The records must be listed in time increasing order.

Path	Type	Dim
/trees/left	H5T_STD_U32LE	N
/trees/right	H5T_STD_U32LE	N
/trees/node	H5T_STD_U32LE	N
/trees/num_children	H5T_STD_U32LE	N
/trees/children	H5T_STD_U32LE	$\leq 2\times$ N

Indexes group

The `/trees/indexes` group records information required to efficiently reconstruct the individual trees from the tree sequence. The `insertion_order` dataset contains the order in which records must be applied and the

`removal_order` dataset the order in which records must be removed for a left-to-right traversal of the trees.

Path	Type
/trees/indexes/insertion_order	H5T_STD_U32LE
/trees/indexes/removal_order	H5T_STD_U32LE

1.6 Citing msprime

If you use `msprime` in your work, please cite the PLoS Computational Biology paper:

Jerome Kelleher, Alison M Etheridge and Gilean McVean (2016), *Efficient Coalescent Simulation and Genealogical Analysis for Large Sample Sizes*, PLoS Comput Biol 12(5): e1004842. doi: 10.1371/journal.pcbi.1004842

Bibtex record:

```
@article{10.1371/journal.pcbi.1004842,
author = {Kelleher, Jerome AND Etheridge, Alison M AND McVean, Gilean},
journal = {PLoS Comput Biol},
title = {Efficient Coalescent Simulation and Genealogical Analysis for Large Sample Sizes},
year = {2016},
month = {05},
volume = {12},
url = {http://dx.doi.org/10.1371%2Fjournal.pcbi.1004842},
pages = {1-22},
number = {5},
doi = {10.1371/journal.pcbi.1004842}
}
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

B

D

G

H

I

L

M

N

P

R

S

T

V

W