`Moments LD` user manual

Corresponding to version 0.0.1

Aaron Ragsdale
Contact: aaron.ragsdale@mail.mcgill.ca

July 12, 2019

Contents

List of Example Codes

1 Introduction to `moments.LD`

Welcome to `moments.LD`, a program for simulating linkage disequilibrium statistics. `moments.LD`, or `mold`, can compute a large set of informative LD statistics for many populations, and performs likelihood-based demographic inference using those statistics.

There are three primary features of `mold` to enable LD-based demographic inference: reading and parsing data, building demographic models, and inferring the parameters of those models by comparing model predictions to data. Typically, we use biallelic SNP data, along with a recombination map, to compute two-locus statistics over a range of genetic distances. We then use `mold` to compute expectations for those statistics under the demographic models we want to test, which can include multiple populations with variable migration, splits and mergers, and population size changes. Using a likelihood-based inference approach, we optimize those models to find the set of parameters that best fit the data.

I've tried to make parsing data and defining demographic models as painless as possible, though the complexity of the program does require some amount of script-writing and interaction. Luckily, `mold` is written in Python, a friendly and powerful programming language. If you are already familiar with ∂a∂i or *moments*, or Python in general, you are in a good position to dive right in to `mold`. If you have limited Python experience, this manual should provide the background and examples to get you up to speed and productive with `mold`.

1.1 Getting help and helping us

Undoubtedly, there will be bugs. If you find a bug in `mold`, or more generally if you find certain aspects of the program to be unintuitive or difficult to use, I would appreciate the feedback. Please submit a bug report at `https://bitbucket.org/simongravel/moments/issues`, and I will try to address the issue in a timely manner. Similarly, if you have suggestions for improved functionality or feature requests, those can be submitted in the issues as well or you can contact me directly.

As we do our own research, *moments* and `mold` are constantly improving. Our philosophy is to include any code we develop for our own projects that may useful to others. If you develop *Moments*-related code that you think might be useful to others, please let me know so I can include it with the main distribution.

2 LD statistics

Patterns of linkage disequilibrium (LD) are informative about evolutionary history, for example for inferring recent admixture events and population size changes or localizing regions of the genome that have experienced recent selective events. LD is commonly measured as the covariance (or correlation) of alleles co-occurring on a haplotype. The covariance (D) is

$$\begin{aligned} D = \mathrm{Cov}(A, B) &= f_{AB} - pq \\ &= f_{AB}f_{ab} - f_{Ab}f_{aB}, \end{aligned}$$

and the correlation (r) is

$$r = \frac{D}{\sqrt{p(1-p)q(1-q)}}.$$

We think of expectations of these quantities as though we average over many realizations of the same evolutionary process, but in reality we have only a single observation for any given pair of SNPs. Therefore in practice we take the averages of LD statistics over many independent pairs of SNPs.

$\mathbb{E}[D]$ is zero genome wide, so LD is often measured by the variance of D ($\mathbb{E}[D^2]$) or the square correlation (r^2), where

$$r^2 = \frac{D^2}{p(1-p)q(1-q)}.$$

Because it is difficult to compute expectations for $\mathbb{E}[r^2]$ under even simple evolutionary scenarios, and because it is difficult to accurately estimate $\widehat{r^2}$ from data, we use $\mathbb{E}[D^2$ and related statistics to compare model predictions for LD to data.

2.1 Hill-Robertson statistics

Hill and Robertson introduced a recursion for $\mathbb{E}[D^2]$ that allows for variable recombination rate between loci and population size changes over time (Hill and Robertson, 1968). To solve for $\mathbb{E}[D^2]$, this system requires additional LD statistics, which we call $Dz = D(1-2p)(1-2q)$ and $\pi_2 = p(1-p)q(1-q)$, where p and q are the allele frequencies at the left and right loci, respectively. This system also relies on heterozygosity (H), so from this system we can compute the vector of statistics

$$y = \begin{pmatrix} \mathbb{E}[D^2] \\ \mathbb{E}[Dz] \\ \mathbb{E}[\pi_2] \\ \mathbb{E}[H] \end{pmatrix}.$$

Instead of computing $\mathbb{E}[r^2]$, which is an expectation of ratios, we use the related statistic $\sigma_D^2 = \frac{\mathbb{E}[D^2]}{\mathbb{E}[\pi_2]}$ (Hill and Robertson, 1968; Ohta and Kimura, 1971). This statistic has the advantage that its expectation can be computed from the Hill-Robertson recursion, and we can accurately compute it from either phased or unphased data.

2.2 Multi-population LD statistics

We extended the Hill-Robertson system to consider LD statistics for multiple populations (Ragsdale and Gravel, 2018). Here, we can model population size changes, splits, mergers, and migration events (pulse or continuous). The statistics we consider take the form

$$\mathbf{z} = \begin{pmatrix} \mathbb{E}[D_i D_j] \\ \mathbb{E}[D_i z_{j,k}] \\ \mathbb{E}[\pi_2(i,j;k,l)] \\ \mathbb{E}[H_{i,j}] \end{pmatrix},$$

where i, j, k, l index populations, and

$$\begin{aligned} D_i z_{j,k} &= D_i(1-2p_j)(1-2q_k), \\ \pi_2(i,j;k,l) &= \frac{1}{4}p_i(1-p_j)q_k(1-q_l) + \frac{1}{4}p_i(1-p_j)q_l(1-q_k) \\ &\quad + \frac{1}{4}p_j(1-p_i)q_k(1-q_l) + \frac{1}{4}p_j(1-p_i)q_l(1-q_k), \\ H_{i,j} &= \frac{1}{2}p_i(1-p_j) + \frac{1}{2}p_j(1-p_i). \end{aligned}$$

From these, we can compare a large number of two-locus statistics. In practice, we work with σ_D^2-like statistics, which are normalized by the π_2 statistic from one of the populations.

3 Getting started

3.1 Downloading `moments` and `mold`

`mold` is packaged and released with `moments`, a python program for running analyses based on the allele frequency spectrum (**?**). `moments` is available at `https://bitbucket.org/simongravel/moments`. Because `mold` continues to be developed and improved, it currently lives on the LD branch of the `moments` repository. For this reason, I recommend cloning the `moments` repository and switching to the LD branch. To do this, in the Terminal navigate to the parent directory where you want to copy `moments` and run

```
git clone https://bitbucket.org/simongravel/moments.git
```

To switch to the LD branch, run

```
git checkout LD
```

Before installing `moments`, we need to ensure that `Python` APR: ...

3.1.1 Dependencies

`moments` and `mold` depend on a number of Python libraries. I strongly recommend using Python 3 (support for Python 2 will be dropped in the near future).

1. Absolute dependencies: `Python`, `numpy`, `scipy`, `cython`, `mpmath`, `pickle`
2. Non-essential dependencies:
   (a) For Plotting, we use `matplotlib` (version $\geq$ 0.98.1)
   (b) For Parsing, we take advantage of `pandas`, `hdf5`, and `scikit-allel` (version $\geq$ 1.2.0) ()
   (c) For Demography building, we use `networkx`

I recommend that you install IPython as well. The easiest way to obtain all these dependencies is to use a package manager, such as Conda (`https://conda.io/en/latest/`) or Enthought (`https://www.enthought.com/product/enthought-python-distribution/`).

To install all required libraries, we can use the requirements.txt file, which lists each of the listed dependencies. If you are using Conda, for example, navigate to the `moments` directory and run
`conda install --file requirements.txt`

3.1.2 Installing

Once the library dependencies are installed, install `moments` and `mold` by running
`python setup.py install`

3.2 Suggested workflow

One of Python's strengths is its interactive nature. When I am first exploring data or writing scripts to build and test models, I often have two windows open: one editing a python script (`script.py`) and the other running an IPython session. That way, I can record my work in the python script and test it as I go. Using IPython, I can call the magic command `%run script.py`, which applies changes I've made in my python script to the IPython session. Note that if I've also changed modules that I've loaded, I'll need to reload those as well. Once I'm happy that I have a usable script, I can call it from the terminal for longer runs of optimization or parsing, using `python script.py`.

Note that we will need to import `moments.LD` to be able to use it: `import moments.LD as mold`.

4 LDstats objects

`mold` represents two-locus statistics using `mold.LDstats` objects, which stores the Hill-Robertson statistics and heterozygosity for one or more populations and for any set of recombination rates. The simplest way to create an `LDstats` object is by defining a set of statistics by hand. For a single population, the order of the LD statistics is $[\mathbb{E}[D^2], \mathbb{E}[Dz], \mathbb{E}[\pi_2]]$ along with heterozygosity $\mathbb{E}[H]$. If the statistics are defined using variables `D2`, `Dz`, `pi2`, and `H`, we would simply call `y = mold.LDstats([[D2, Dz, pi2], [H]], num_pops = 1)`. For example, if we set `D2, Dz, pi2, H = 1e-7, 1e-8, 2e-7, 1e-3` in this example, and then if we print `y`, we would see as the output:

```
Out[1]: LDstats([[1.e-07 1.e-08 2.e-07]], [0.001], num_pops=1, pop_ids=None)
```

To see which statistics each value corresponds to, we can call `y.names()`, which would output:

```
Out[2]: (['DD_1_1', 'Dz_1_1_1', 'pi2_1_1_1_1'], ['H_1_1'])
```

Typically, we either compute the `LDstats` from a demographic model, or we build the object from data. We'll walk through both in later sections. First, we'll use build in model functions to explore what information is stored in `LDstats` objects, and how to manipulate the objects.

To obtain the expected statistics at equilibrium (steady-state demography), we can call `mold.Demographics1D.snm` (snm stands for standard neutral model). We can specify the per-base population size-scaled mutation rate $\theta = 4N_{\text{ref}}\mu$ (default set to $\theta = 0.001$) and population size-scaled recombination rates separating loci $\rho = 4N_{\text{ref}}r$

(default set to `None`). `mold` can handle any number of recombination rates (zero, one, or multiple), and the returned `LDstats` object will contain LD statistics for as many recombination rates as you gave it.

To give some examples,

```
In [3]: mold.Demographics1D.snm()
Out[3]: LDstats([], [0.001], num_pops=1, pop_ids=None)

In [4]: mold.Demographics1D.snm(rho=0)
Out[4]: LDstats([[1.38888889e-07 1.11111111e-07 3.05555556e-07]], [0.001], num_pops=1, pop_ids=None
)

In [5]: mold.Demographics1D.snm(rho=[0,1,2,10], theta=0.01)
Out[5]:
LDstats([[1.38888889e-05 1.11111111e-05 3.05555556e-05]
[8.59375000e-06 6.25000000e-06 2.81250000e-05]
[6.25000000e-06 4.16666667e-06 2.70833333e-05]
[2.01612903e-06 8.06451613e-07 2.54032258e-05]], [0.01], num_pops=1, pop_ids=None)
```

In this last example, the four sets of LD statistics correspond to $\rho = 0, 1, 2$, and 10, respectively, while expected heterozygosity is only shown a single time ($\mathbb{E}[H] = 0.01$). In each case, `num_pops` is automatically set to `1`, and because we didn't specify population IDs, `pop_ids = None`. `y.LD()` will return just the LD statistics, while `y.H()` returns just the heterozygosity statistics.

5 Parsing and importing data

`mold` can import data and compute LD statistics from either phased or unphased sequencing data. Typically, data is stored in a VCF formatted file. We parse the VCF using `scikit-allel` to get genotype arrays, which we then iterate over to count two locus haplotype (phased data) or genotype (unphased data) frequencies for pairs of SNPs. We then use two-locus haplotype or genotype counts to compute statistics in the multi-population Hill-Robertson basis. If we are binning data based on recombination distance, we will also need a recombination map, and if there are multiple populations, we will need to provide a file that identifies individuals with their population (formats for each are described below). If we are interested in data from a subset of the full data (say we want to keep only intergenic variants, or want to focus on a particular region), we can provide a bed file that defines the regions or features we should parse.

5.1 Computing statistics from genotype arrays

To start simply, we might be interested in computing pairwise LD statistics for a given set of genotypes or between two sets of genotypes. A genotype array G has size $L \times n$, where L is the number of SNPs and n is the number of sequenced diploid individuals, so that entry (i, j) is the genotype state of individual j at SNP i. Genotype states are either 0 (homozygous reference), 1 (heterozygous), or 2 (homozygous alternate). `mold.Parsing` counts and computes statistics from genotype arrays using the approach described in (Ragsdale and Gravel, 2019).

To get pairwise statistics for each possible pair (all $L(L-1)/2$ of them), use
`mold.Parsing.compute_pairwise_stats(G)`,
where `G` is the genotype matrix as described above. This will output three vectors, each of length $L(L-1)/2$, for D^2, Dz, and π_2, in that order. To convert to a (symmetric) $L \times L$ matrix of pairwise statistics, we can use `numpy`'s `triu` function.

To get the average Hill-Robertson statistics over all pairwise comparisons for a block of SNPs, use
`mold.Parsing.compute_average_stats(G)`,
which returns the mean values of D^2, Dz, and π_2, in that order.

To compute Hill-Robertson statistics between two sets of genotype data, stored in two genotype arrays `G1` and `G2`, we can similarly call
`mold.Parsing.compute_pairwise_stats_between(G1, G2)`
(which returns vectors of size $L_1 L_2$, where L_i is the number of SNPs in genotype array G_i) and
`mold.Parsing.compute_average_stats_between(G1, G2)`.

5.2 Computing statistics from a VCF

In our analyses, we are interested in computing two-locus statistics from genotype data stored in a VCF for pairs of SNPs at varying genetic distances. To parse a VCF file and output Hill-Robertson statistics, we use `mold.Parsing.compute_ld_statistics`. The only required input for this function is the VCF filename. Otherwise, there are a number of options and arguments that can be passed to this function.

- `bed_file` : default set to `None`. To only parse variants in given regions, we specify those regions in a bed file.
- `rec_map_file` : default set to `None`. If we pass a recombination map filename, we can specify the formatting of the recombination map file (see subsection below).
- `pop_file` : default set to `None`. If `None`, it uses data from all individuals as a single population.
- `pops` : If we pass a population file, we can specify which populations to parse using `pops=[pop1, pop2, ...]`.
- `bp_bins` : default set to `None`. If we do not pass a recombination map, we will parse the data based on base pair distance between SNPs. Here, we pass a list of bin edges. For example, to parse data into bins of $(0, 10$ kb$]$, $(10$ kb$, 20$ kb$]$, and $(20$ kb$, 30$ kb$]$, we would set `bp_bins = [0, 10000, 20000, 30000]`.
- `use_genotypes` : default set to `True`, which we used with unphased data. Set to `False` for phased data.
- `stats_to_compute` : default set to `None`. If `None`, computes all statistics in the multi-population H-R basis. We can also specify just a subset of these statistics, if desired.
- `report` : default set to `True`. If `True`, outputs progress report as it parses the VCF file.

In the simplest case without a recombination map, we can compute statistics from a VCF file based on physical (bp) distance. To do this, for a single population, we only need to specify the VCF filename and bin edges in base pairs:

```
ld_stats = mold.Parsing.compute_ld_statistics('path/to/vcf/file.vcf.gz', bp_bins=[0, 10000, 20000,
 30000]).
```

5.3 Multiple populations

To parse data for multiple populations, we need to also include a file that tells us which population each individual belongs to. In the population file, each row corresponds to an individual in the VCF file (individual names must match to those labeled in the VCF header). In each row, the first column is the individual ID, and the second column is the population name. Additional columns are ignored. We could then call

```
ld_stats = mold.Parsing.compute_ld_statistics('path/to/vcf/file.vcf.gz', pop_file='path/to/pop/file
.txt', pops=[pop1, pop2, pop3], bp_bins=[0, 10000, 20000, 30000]).
```

5.4 Using a recombination map

Because recombination rates can vary along the genome, we often want to parse two-locus data by the genetic instead of physical distance separating SNPs. To use a recombination map to parse data, we specify the file path and name using `rec_map_file`. In the recombination map file, the first column corresponds to physical positions, and other columns correspond to the genetic position in either Morgans or cM. If there are multiple maps in the file, we can specify the map we want to use by the map name in the header.

For example, the first and last few lines of a set of recombination maps of chromosome 22 for human reference genome build hg19 (available here: `https://www.well.ox.ac.uk/~anjali/AAmap/`) are

```
"Physical_Pos""deCODE""COMBINED_LD""YRI_LD""CEU_LD""AA_Map""African_Enriched""Shared_Map"
16051347 0 0 0 0 0 0 0
16052618 0 0.0099 0.0083 0.0163 0 0 0
16053624 0 0.0177 0.0148 0.0293 0 0 0
16053659 0 0.0179 0.0151 0.0297 0 0 0
16053758 0 0.0187 0.0157 0.031 0 0 0
...
51217134 55.5922 73.6005 75.6968 72.5384 68.9516 67.8206 54.8248
51219006 55.5922 73.6017 75.6982 72.5398 68.9516 67.8206 54.8248
```

```
51222100 55.5922 73.6037 75.7006 72.5421 68.9516 67.8206 54.8248
51223637 55.5922 73.6047 75.7018 72.5433 68.9516 67.8206 54.8248
51229805 55.5922 73.6088 75.7068 72.5479 68.9516 67.8206 54.8248
```

Cumulative distances are given in cM. If we want to use the (Hinch et al., 2011) African American admixture map ("AA_Map") from this file, setting recombination bins using `r_bins` as, for example,

```
r_bins = [0, 1e-5, 2e-5, 5e-5, 1e-4, 2e-4, 1e-3]
```

we call:

```
ld_stats = mold.Parsing.compute_ld_statistics('path/to/vcf/file.vcf.gz', rec_map_file='path/to/rec
/map/file', map_name='AA_Map', map_sep='', cM=True, r_bins=r_bins).
```

`map_sep` by default is set to tab separation, so we'll need tell the parsing function that this map is separated by spaces.

5.5 Parsing example

APR: Explain `msprime_two_pop_parsing.py`

5.6 Creating bootstrap datasets

5.6.1 Computing sets of LD statistics

5.6.2 Computing statistic averages and covariances

6 Specifying a model

In `mold`, we want to build and test demographic models by computing LD statistics for a specified model. `mold` allows two ways to specify demographic models, either through direct manipulation of `LDstats` objects, or by defining a demography through a graphical structure. In this section, we'll describe and give examples for each. We'll start with the direct manipulation of `LDstats` objects to give intuition about how `mold` computes LD statistics under different demographic scenarios. For more than just a couple populations, it is far easier to implement models using the Demography module, which builds the demography from a user-defined directed graph.

6.1 Implementation

Implementation should be familiar to ∂a∂i and `moments` users. Once we have defined a `LDstats` object, we can perform demographic functions and manipulations on it and integrate it forward in time. We can start by specifying the initial distribution, before applying demographic events:

```
y = mold.Demographics1D.snm(rho = [0,1,10], theta=0.001).
```

`y` is now the single population steady-state set of LD statistics for the specified ρ and θ parameters.

From here, we can integrate forward in time with a specified relative population size, or we can split the population into two daugher populations. To integrate forward in time, with a single population, we call:

```
y.integrate([nu], T, rho=[0,1,10], theta=0.001),
```

where `nu` is a relative population size and `T` is the integration time (in genetic units). To split into two, we call:

```
y = y.split(1).
```

In the `split` function, we specify the population number we want to split, and a new population is appended to the end of the population list. For example, if `y` currently has two populations, and we want to split population 1 into 1A and 1B, we call `y.split(1)`, which returns a new `LDstats` object with populations ordered as (1A, 2, 1B).

Below, I write out some examples for defining demographic models (a bottleneck model with recent growth, an isolation with migration model, and the Gutenkunst out-of-Africa model).

6.2 The `Demography` builder

Manual specification of demographic functions becomes increasingly difficult as the number of populations in the model grows. With more than two or three populations, events that occur along different branches or lineages can switch their order, so that writing a flexible demographic model requires a host of pesky, bug-prone, if-then statements. `mold` introduces a more user-friendly method to define demographic models through the `Demography` module. `mold.Demography` uses `networkx` to represent demographic models as directed acyclic graphs, where nodes specify populations with attributes (such as size functions, migration rates, frozen ancient sample, etc), and edges define relationships between populations.

To start, we `import networkx as nx` and define an empty graph as `G = nx.DiGraph()`, which will store nodes (populations) and edges (relationships). To add a population, we use `G.add_node`, and specify the population name, the population size or size function, and the time the population persists, either up to the simulation ending, splitting or transitioning to other population(s), or extinction some time before present. We can also specify (optional) migration rates *from* this population *to* other populations. To specify migration from other populations to this population, we add migration rates to that other population.

6.2.1 Example: a two-population IM model

For example, let's set up a model where a single population doubles in size for time 0.1 before splitting into two populations (pop0 and pop1). These split populations have different relative population size (say 3.0 and 0.5) and they remain split for time 0.2 with symmetric migration rate 1.0.

We first set the initial 'root' population, which starts at equilibrium (set relative size $\nu = 1$ and $T = 0$):
`G.add_node('root', nu=1.0, T=0)`
We name the pre-split population 'pop0', and set
`G.add_node('pop0', nu=2.0, T=0.1)`
We add the two split populations, along with migration rates, as
`G.add_node('pop1', nu=3.0, T=0.2, m={'pop2': 1.0})`
and
`G.add_node('pop2', nu=0.5, T=0.2, m={'pop1': 1.0})`

Finally, we set the graph edges, which defines the topology of the demography:
`G.add_edges_from([('root', 'pop0'), ('pop0', 'pop1'), ('pop0', 'pop2')])`
A similar model (without the pre-split population size change) is given in Example Code 3.

To simulate LD statistics on this demography for a given set of recombination and mutation rates, we call
`y = mold.Demography.evolve(G, rho=[0,1], theta=0.001, pop_ids=['pop1', 'pop2'])`
where `pop_ids` specifies the order we would like the statistics to be output.

6.3 Units

`mold`, like `moments` and ∂a∂i, uses genetic units instead of physical units to define models. Time and rates are typically measured in or scaled by $2N_{\text{ref}}$:

- Time is given by units of $2N_{\text{ref}}$ generations.
- The mutation rate is $\theta = 4N_{\text{ref}}u$, where u is the per-base per-generation mutation rate.
- Migration rates are $2N_{\text{ref}}m_{ij}$, where m_{ij} is the per lineage migration rate from population i to population j. In other words, m_{ij} is the probability that any given lineage in population j had its parent in population i in the previous generation.
- The recombination rate is $\rho = 4N_{\text{ref}}r$, where r is the probability of a recombination event occurring between two loci in a lineage in one generation.

7 Example Code

In this section, we give some example demographic functions. Some are specified by initializing the equilibrium `LDstats` object, and then applying demographic events such as size changes and splits, and integrating forward in time. Others are specified using the `Demography` module. Here, we use `networkx` to define a population topology and attributes for each population, and then we call `mold.Demography.evolve` to compute the expected statistics.

In the directory moments/examples/LD, you will find additional example code, including a demographic model for the out of Africa model augmented by Neanderthal introgression into the Eurasian population and a deep split and subsequent migration with an archaic population within Africa (similar to the demographic model inferred in (Ragsdale and Gravel, 2019). Also in the examples directory, we use a subset of populations from the publicly available Simons Diversity Project (Mbuti, Punjabi, Dai, and Papuan) (**?**) along with high coverage Neanderthal and Denisovan individuals (**?**) to parse LD statistics. We then fit a demographic model with recent events (modern human splits, size changes, and migration rates) along with the timing of splits between modern humans, Neanderthal and Denisovans, and a hypothesized archaic African population, the split between Neanderthal and Denisovan populations, and the timing and magnitude of admixture from archaic populations into human lineages. The topology of this model is shown in Figure **??**.

```
def bottleneck_growth(params, rho=None, theta=0.001):
    """
    Instantaneous bottleneck to size nuB
    followed by exponential growth to size nuF over time T
    """
    nuB, nuF, T = params
    nu_func = lambda t: [nuB * np.exp(np.log(nuF/nuB)
                           * t / T)]

    y = mold.Demographics1D.snm(rho=rho, theta=theta)
    y.integrate(nu_func, T, rho=rho, theta=theta)

    return y
```

Example Code 1: **Bottleneck model:** At time `T` in the past, an equilibrium population goes through a bottleneck of depth `nuB`, recovering to relative size `nuF` through exponential growth. In all examples listed here, we need to `import numpy as np` and `import moments.LD as mold`.

```
def IM(params, rho=None, theta=0.001):
    """
    Population split T generations ago
    with relative sizes nu1 and nu2, and symmetric
    migration rates m
    """
    nu1, nu2, T, m = params

    y = mold.Demographics1D.snm(rho=rho, theta=theta)
    y = y.split(1)
    y.integrate([nu1, nu2], T, m=[[0,m],[m,0]],
        rho=rho, theta=theta)

    return y
```

Example Code 2: **IM model:** One population splits into two some time in the past. Each population can have a new size, with symmetric and continuous migration between populations.

```
def IM_graph(params, rho=None, theta=0.001, pop_ids=['pop1', 'pop2']):
    """
    Population split T generations ago
    with relative sizes nu1 and nu2, and symmetric
    migration rates m
    """
    nu1, nu2, T, m = params

    G = nx.DiGraph()
    G.add_node('root', nu=1.0, T=0)
    G.add_node('pop1', nu=nu1, T=T, m={'pop2': m})
    G.add_node('pop2', nu=nu2, T=T, m={'pop1': m})

    G.add_edges_from([('root', 'pop1'), ('root', 'pop2')])

    y = mold.Demography.evolve(G, theta=theta,
        rho=rho, pop_ids=pop_ids)

    return y
```

Example Code 3: **IM model using Demography:** The same isolation with migration model, defined using the graphical representation of the Demography method.

```
def OutOfAfrica(params, rho=None, theta=0.001):
    """
    The 13 parameter out of Africa model introduced in
    Gutenkunst et al. (2009)
    """
    (nuA, TA, nuB, TB, nuEu0, nuEuF, nuAs0,
        nuAsF, TF, mAfB, mAfEu, mAfAs, mEuAs) = params

    y = mold.Demographics1D.snm(rho=rho, theta=theta)
    y.integrate([nuA], TA, rho=rho, theta=theta)

    y = y.split(1)
    mig_mat = [[0,mAfB], [mAfB,0]]
    y.integrate([nuA, nuB], TB, m = mig_mat,
        rho=rho, theta=theta)

    y = y.split(2)
    nu_func = lambda t: [nuA,
        nuEu0 * np.exp(np.log(nuEuF/nuEu0) * t/TF),
        nuAs0 * np.exp(np.log(nuAsF/nuAs0) * t/TF)]
    mig_mat = [[0, mAfEu, mAfAs],
        [mAfEu, 0, mEuAs],
        [mAfAs, mEuAs, 0]]
    y.integrate(nu_func, TF, m = mig_mat,
        rho=rho, theta=theta)

    return y
```

Example Code 4: **Out of Africa model:** The Gutenkunst Out-of-Africa model (Gutenkunst et al., 2009), with 13 parameters as originally defined. This model has three representative continental populations (often YRI, CEU, and CHB), with an out of Africa split between Eurasian and African populations, followed by a split between European and East Asian populations, with symmetric migration rates and size changes along each branch.

```
def OutOfAfrica_graph(params, rho=None, theta=0.001,
        pop_ids=['YRI', 'CEU', 'CHB']):
    (nuA, TA, nuB, TB, nuEu0, nuEuF, nuAs0,
        nuAsF, TF, mAfB, mAfEu, mAfAs, mEuAs) = params

    G = nx.DiGraph()

    # add the population nodes, with sizes, times, and migrations
    G.add_node('root', nu=1, T=0)
    G.add_node('A', nu=nuA, T=TA)
    G.add_node('B', nu=nuB, T=TB,
        m={'YRI': mAfB})
    G.add_node('YRI', nu=nuA, T=TB+TF,
        m={'B': mAfB, 'CEU': mAfEu, 'CHB': mAfAs})
    nu_func_Eu = lambda t: nuEu0 * np.exp(np.log(nuEuF/nuEu0) * t/TF)
    G.add_node('CEU', nu=nu_func_Eu, T=TF,
        m={'YRI': mAfEu, 'CHB': mEuAs})
    nu_func_As = lambda t: nuAs0 * np.exp(np.log(nuAsF/nuAs0) * t/TF)
    G.add_node('CHB', nu=nu_func_As, T=TF,
        m={'YRI': mAfAs, 'CEU': mEuAs})

    # define topology of population graph
    G.add_edges_from([('root', 'A'), ('A', 'YRI'), ('A', 'B'),
        ('B', 'CEU'), ('B', 'CHB')])

    # evolve using Demography.evolve
    y = mold.Demography.evolve(G, theta=theta,
        rho=rho, pop_ids=pop_ids)

    return y
```

Example Code 5: **Out of Africa Demography graph:** The same model as Example Code 4, but defined using the Demography module. The Demography method takes advantage of the package `networkx`, which we `import networkx as nx`. Here, we define populations (nodes) with attributes (such as migration rates and sizes), and then define edges to relate populations.

8 Simulation and Inference

8.1 Comparing model to data

In the Section Specifying the model, we showed two approaches to defining demographic models. `mold` computes the Hill-Robertson statistics ($\mathbb{E}[D_i^2]$, $\mathbb{E}[D_iD_j]$, etc), which give expectations for any pair of loci separated by a given recombination rate. When running inference, we prefer to use statistics normalized by the joint heterozygosity (π_2) of one of the populations, by default the first population in `pop_ids` (as in (Rogers, 2014) and (Ragsdale and Gravel, 2018)). This has the advantage of removing dependence of the statistics on the underlying mutation rate.

By using the ratio $\mathbb{E}[\text{stat.}]/\mathbb{E}[\pi_2]$, it also makes computing statistics from data much simpler, as we don't need to computing the total number of pairs per recombination bin. Instead we just sum D^2, Dz, and π_2 over all pairs of SNPs within a bin, and then divide by the sum of all contributions to π_2 for the same bin. This gives σ_D^2-type statistics for all terms in the multi-population Hill-Robertson basis. As described above in Section 5.6, `mold.Parsing.bootstrap_data` computes average statistics and their covariances, after parsing data over subregions of the genome using `mold.Parsing.compute_ld_statistics`.

APR: To compute model expectations for these same statistics, normalized by a given population's $\mathbb{E}[\pi_2]$ and $\mathbb{E}[H]$, we use our demographic function with the wrapper function `mold.Inference.wrap_sigmaD2`, which takes the same arguments as our demographic function, as well as the index of the population to normalize by. If our demographic function is `OutOfAfrica_graph`, which takes the 13 demographic parameters, ρ, θ, and `pop_ids`, we compute normalized statistics by

```
y = mold.Inference.sigmaD2(OutOfAfrica_graph, ...
```

APR: `mold.Inference.sigmaD2` converts to normalized statistics

APR: To get σ_D^2 statistics for bins, we can pass the demographic function, bin edges `rho=[rho0, rho1, ...]`, θ, and `pop_ids` to `mold.Inference.bin_stats`, which computes expected statistics for a bin using the Simpson's rule.

APR: best way to take a demographic function and return σ_D^2 type statistics, or expectations over bins using Simpson's rule, etc...? Build into Inference.

To visually compare data and model predictions, using `mold.Plotting`, described in Section 9.

8.1.1 Likelihoods

To compare model predictions to observed data, we use a likelihood approach. `mold` estimates composite likelihoods using a multivariate Gaussian for each bin. For a given bin, we assume that we have computed estimates for the average statistics $\hat{\mathbf{x}}_{(\rho_0,\rho_1)}$ and the covariance matrix of those statistics from data $\Sigma_{(\rho_0,\rho_1)}$, as well as the model prediction for those statistics for the known recombination rate bin $\mathbf{y}_{(\rho_0,\rho_1)}$. Then the log-likelihood of the model parameters Θ for that bin is given by

$$\mathcal{L}(\Theta|\hat{\mathbf{x}}_{(\rho_0,\rho_1)}) = \mathcal{N}\left(\hat{\mathbf{x}}_{(\rho_0,\rho_1)}|\mathbf{y}_{(\rho_0,\rho_1)}, \Sigma_{(\rho_0,\rho_1)}\right).$$

This log-likelihood is computed by calling `mold.Inference.ll(x, y, sigma)`, where `x` is the data, `y` is the model prediction, and `sigma` is the covariance matrix.

To compute the composite likelihood over multiple bins, we simply approximate it as the product of likelihoods of each bin. This is computed using `mold.Inference.ll_over_bins(xs, mus, sigmas)`, where `xs`, `mus`, and `sigmas` are lists of the data, model predictions, and covariance matrices for each bin.

8.2 Fitting

APR: above need to discuss that we typically also infer N_e based on the recombination map, since they are typically given in raw recombination rates

For observed data, the goal here is to propose a model and find the parameters of the model that best fit the data. We use functions in `mold.Inference` to optimize model parameters, given computed average statistics for each recombination bin and the associated covariance matrices. We take data that has been parsed over n recombination

bins, $\{(r_0, r_1], (r_1, r_2], \ldots, (r_{n-1}, r_n]\}$, and use the optimization functions in `Inference` to explore parameter space of our model (here, called `model_func`) to find the optimal parameter values.

The most common usage of the optimization functions requires the following input.
Required inputs:

- The initial parameter guess `p0` : this is the list of model parameters, and it is typically augmented by the reference N_e, which is used to scale raw recombination rates (r) to get ρ values. APR: with or without N_e on the end
- `data` as two lists. The first list are the mean statistics, which has size $n + 1$. The first n entries of the list of means are statistic arrays for each bin (sorted in the order of recombination bins), and the last entry in the list is the set of heterozygosity statistics APR: as output by `Parsing` - does it prefilter the normalized statistic out? or pass all statistics as well as an option for telling it which statistic you normalized by?. The second list in `data` are the corresponding covariance matrices.
- `model_func`, which computes (unnormalized) LD statistics APR: in form of Example Codes
- `rs` : the list of raw recombination rate bin edges, such as $r_{\text{edges}} = [r_0, r_1, \ldots, r_n]$. If we use `rs`, we set `Ne` to a fixed value of N_{ref} to scale recombination rates, or we use the last entry in the list of parameters, in which case N_{ref} is a parameter to be fit.

Optional inputs:

- `normalization` : The population used to normalize σ_D^2 statistics. Default set population 1, which uses $\pi_2(1)$ and $H(1)$ statistic in the first population.
- `verbose` : Set to `True` if we want to output updates of function optimization (integer values tell how often to output updates)
- `fixed_params` : A list the same length as `p0`. Default set to `[None]*len(p0)`. For any values to be fixed (and not optimized over), set that position to the fixed value.
- `upper_bounds` and `lower_bounds` : Parameters can sometimes diverge to unrealistic values during optimization. To constrain parameter values to a given interval, use `upper_bounds` and `lower_bounds`, in the same way as `fixed_params`.

8.2.1 Optimization functions

APR: log fmin and powell

Suppose I have a list of statistics means `ms` and covariances `vs` over bins defined by bin edges `r_bins`, and a model I wish to optimize `model_func` that takes parameters `[p1,p2,...,pn]`. I set my initial guess `p0 = [guess_p1, guess_p2, ..., guess_pn, guess_Ne]`. To run optimization, I call
`mold.Inference.optimize_log_fmin(p0, [ms, vs], [model_func], rs=r_bins, verbose=1)`.

8.3 Uncertainty analysis

APR: to come

(Coffman et al., 2016)

9 Plotting

APR: Under development

9.1 Visualizing LD curves

9.2 Residuals

10 The full two-locus frequency spectrum

10.1 `moments.TwoLocus`

10.1.1 Specifying models

10.1.2 Parameters

10.1.3 Selection

11 Frequently asked questions

1. What if I'm having issues running or installing this program?

   Bug: issues

   Bigger issues or difficulties: email

2. How do I cite `mold`?

12 Acknowledgements

`moments` was originally developed by Julien Jouganous, and based off of Ryan Gutenkunst's ∂a∂i software. `moments` and `mold` (and indeed much of my own scientific development) are greatly indebted to Ryan Gutenkunst. Not only are these programs modeled off of ∂a∂i's interface and functionality, but also my general approach to writing, testing, and using scientific software has been guided and influenced by Ryan.

References

Coffman, A. J., Hsieh, P. H., Gravel, S., and Gutenkunst, R. N. (2016). Computationally Efficient Composite Likelihood Statistics for Demographic Inference. *Molecular Biology and Evolution*, 33(2):591–593.

Gutenkunst, R. N., Hernandez, R. D., Williamson, S. H., and Bustamante, C. D. (2009.). Inferring the joint demographic history of multiple populations from multidimensional SNP frequency data. *PLoS Genetics*, 5(10):e1000695.

Hill, W. G. and Robertson, A. (1968.). Linkage disequilibrium in finite populations. *Theoretical and Applied Genetics*, 38(6):226–231.

Hinch, A. G., Tandon, A., Patterson, N., Song, Y., Rohland, N., Palmer, C. D., Chen, G. K., Wang, K., Buxbaum, S. G., Akylbekova, E. L., Aldrich, M. C., Ambrosone, C. B., Amos, C., Bandera, E. V., Berndt, S. I., Bernstein, L., Blot, W. J., Bock, C. H., Boerwinkle, E., Cai, Q., Caporaso, N., Casey, G., Cupples, L. A., Deming, S. L., Diver, W. R., Divers, J., Fornage, M., Gillanders, E. M., Glessner, J., Harris, C. C., Hu, J. J., Ingles, S. A., Isaacs, W., John, E. M., Kao, W. H., Keating, B., Kittles, R. A., Kolonel, L. N., Larkin, E., Marchand, L. L., McNeill, L. H., Millikan, R. C., Murphy, A., Musani, S., Neslund-Dudas, C., Nyante, S., Papanicolaou, G. J., Press, M. F., Psaty, B. M., Reiner, A. P., Rich, S. S., Rodriguez-Gil, J. L., Rotter, J. I., Rybicki, B. A., Schwartz, A. G., Signorello, L. B., Spitz, M., Strom, S. S., Thun, M. J., Tucker, M. A., Wang, Z., Wiencke, J. K., Witte, J. S., Wrensch, M., Wu, X., Yamamura, Y., Zanetti, K. A., Zheng, W., Ziegler, R. G., Zhu, X., Redline, S., Hirschhorn, J. N., Henderson, B. E., Taylor, H. A., Price, A. L., Hakonarson, H., Chanock, S. J., Haiman, C. A., Wilson, J. G., Reich, D., and Myers, S. R. (2011.). The landscape of recombination in African Americans. *Nature*, 476(7359):170–175.

Ohta, T. and Kimura, M. (1971.). Linkage disequilibrium between two segregating nucleotide sites under the steady flux of mutations in a finite population. *Genetics*, 68(4):571–580.

Ragsdale, A. P. and Gravel, S. (2018.). Models of archaic admixture and recent history from two-locus statistics. *BioRxiv*, doi:10.1101/489401.

Ragsdale, A. P. and Gravel, S. (2019.). Unbiased estimation of linkage disequilibrium from unphased data. *BioRxiv*, doi:10.1101/557488.

Rogers, A. R. (2014.). How population growth affects linkage disequilibrium. *Genetics*, 197(4):1329–1341.