moments

Release 1.1.0

Aaron Ragsdale

Apr 06, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Note: These docs are under development. In particular, many of the modules have not yet been completed and some of the extensions are not documented in great detail. If you find any issues, confusing bits, or have suggestions to make them more complete or clearer, please open an issue or a PR. Thanks!

Welcome to `moments`! `moments` implements methods for inferring demographic history and patterns of selection from genetic data, based on solutions to the diffusion approximations to the site-frequency spectrum (SFS). The SFS implementation and interface of `moments` is large based on the ai open source package developed by Ryan Gutenkunst. We largely reuse `ai`'s interface but introduced a new simulation engine. This new method is based on the direct computation of the frequency spectrum without solving the diffusion system. Consequently we circumvent the numerical PDE approximations and we get rid of the frequency grids used in `ai`.

`moments.LD` implements methods for computing linkage disequilibrium statistics and running multi-population demographic inference using patterns of LD. This extension contains methods for parsing phased or unphased sequencing data to compute LD-decay for a large number of informative two-locus statistics, and then uses those statistics to infer demographic history for large numbers of populations.

`moments` was developed in Simon Gravel's group in the Human Genetics department at McGill University, with maintenance and development by the Gravel Lab and Aaron Ragsdale.

1.1 Citations

If you use `moments` in your research, please cite:

- Jouganous, J., Long, W., Ragsdale, A. P., & Gravel, S. (2017). Inferring the joint demographic history of multiple populations: beyond the diffusion approximation. *Genetics*, 206(3), 1549-1567.

If you use `moments.LD` in your research, please cite:

- Ragsdale, A. P. & Gravel, S. (2019). Models of archaic admixture and recent history from two-locus statistics. *PLoS Genetics*, 15(6), e1008204.
- Ragsdale, A. P. & Gravel, S. (2020). Unbiased estimation of linkage disequilibrium from unphased data. *Mol Biol Evol*, 37(3), 923-932.

If you use `moments.TwoLocus` in your research, please cite:

- Ragsdale, A. P. (2022). Local fitness and epistatic effects lead to distinct patterns of linkage disequilibrium in protein-coding genes. *Genetics*, 221(4), iyac097.

1.2 Change log

1.2.1 1.1.15

- Fix various bugs in LD parsing methods, including when data is missing and recursion errors in cythonized genotype calculation methods
- Add steady state solution to LD methods

1.2.2 1.1.14

- Fix bugs when computing multi-population LD statistics using phased haplotype data
- Steady state LD statistics for two-population island models

1.2.3 1.1.13

- Function to parse ANGSD-formatted data as a moments.Spectrum object (issue #106)
- Catch if genotype matrix is too large to compute pairwise LD (issue #105)

1.2.4 1.1.12

- Efficiency improvements in LD Parsing and Integration
- Test demes graph slicing features

1.2.5 1.1.11

- The LD inference methods now allow calculation of f-statistics (f2, f3, f4)
- Demes methods allow multiple sources in pulses
- Demes integration allow for ancient samples
- Fix bugs in L-BFGS-B methods for inference using the SFS

1.2.6 1.1.10

- Add warnings and exceptions if bins are improperly defined in LD.Parsing (Issue #99).
- Remove `ld_extensions` flag from installation so that all extensions are built automatically.
- Pin cython to ~0.29 until recursion error is fixed
- Allow samples to be specified with a dictionary for SFS calculation with Demes
- Memory-efficient caching of projection in TwoLocus
- Add LD inference using Demes and clean up uncertainty calculations for SFS inference using demes

1.2.7 1.1.9

- Allow ancient samples in Demes inference function
- Add selection and dominance to Demes SFS integration function
- Add f2 and f4 statistics to LDstats object
- Allow multiple simultaneous merger events in Demes integration methods
- Add uncertainty functions to Demes SFS inference module
- Refactor Demes SFS inference options (#85)
- Add function to compute genotype matrix from the SFS
- Add function to compute allele frequency threshold LD statistics from TwoLocus spectrum
- Fix factor of 2 discrepancy between LD and TwoLocus mutation model (#60)

1.2.8 1.1.8

- Fix bug that plotted multiple colorbars in plot_single_2d_sfs (issue #82).
- Add L-BFGS-B optimization method to LD inference.
- Fix bug in SFS inference using demes when a branch event time is a variable parameter.
- Fix bug in LD Godambe method that improperly normalized J matrix and cU vector.

1.2.9 1.1.7

- Inference using demes allows for ancestral misidentification estimation (#81).
- Fst computation now has option for all pairwise computations (#80).
- Bug fix when computing LD with an input VCF that includes multiple chromosomes (#78).
- Bug fix when computing LD means over multiple regions.
- Expanded documentation, particularly for clarification of installation steps in docs when using LD parsing methods (#79), usage of Godambe methods for computing confidence intervals (#77), and more details for LD methods.

1.2.10 1.1.6

- Many small bug fixes and API improvements to LD parsing, inference, and confidence interval methods.
- Expanded documentation for computing, parsing, and running inference using LD statistics (#73).
- Expand LD examples in repository and bring them up to date with current API (#74).
- Minor improvements to 1D SFS plotting (#64).

1.2.11 1.1.5

- Use (chrom, pos) tuple as data dictionary key, to avoid conflicts with underscores. Underscores in contig/chromosome names are again supported.
- Add branch function to Spectrum class.
- Fix bug when computing SFS from demes with branches occurring simultaneously (#71).
- Fix bug when computing SFS from demes with pulses occurring simultaneously (#72).

1.2.12 1.1.4

- Fix bugs in Plotting multi-population SFS comparisons that were showing each subplot in a new figure instead of in a single plot.
- Hide the intrusive scale bar in ModelPlot by default.

1.2.13 1.1.3

- Fix bug in Misc.make_data_dict_vcf that skipped any site with missing data.
- Fix numpy deprecation warning when projecting.
- Documentation updates for miscellaneous functions.
- Fix bug where copying and pickling LDstats objects resulted in a recursion error (#66).

1.2.14 1.1.2

- Fix bug when checking if matplotlib is installed for model plotting (issue #68).
- Now compatible with demes >= 0.1.

1.2.15 1.1.1

- Fix a pesky RecursionError in `moments.LD.Inference.sigmaD2`.
- Fix bug when simulating LD using `Demes` if admixture timing coincides with a deme's end time.
- Fix `numpy.float` deprecation warning in `moments.LD.Numerics`.
- Update demes methods to work with `demes` version 0.1.0a4.
- Improve (or at least change) some of the plotting outputs.
- Protect import of `demes` if not installed.

1.2.16 1.1.0

- Completely rebuilt documentation, now hosted on [Read the Docs](https://moments.readthedocs.io/).
- Tutorials and modules in the documentation for running inference, inferring the DFE, and exploring LD under a range of selection models.
- More helpful documentation in docstrings.
- Support for [demes](https://moments.readthedocs.io/en/latest/extensions/demes.html).
- Simpler functions to improve Spectrum manipulation and demographic events, such as fs.split(), fs.admix, etc.
- API and numerics overhaul for Triallele and TwoLocus methods.
- Expanded selection models in the TwoLocus module.
- moments.LD methods are now zero-based.
- Reversible mutation model supports a single symmetric mutation rate.

1.2.17 1.0.9

- Numpy version bump from 0.19 to 0.20 creates incompatibility if cython extension are built with different version than user environment. This more explicitly specifies the numpy version to maintain compatibility (with thanks to Graham Gower).

1.2.18 1.0.8

- Allow for variable migration rate by passing a function as the migration matrix (with thanks to Ekaterina Noskova/@noscode).
- Fixes an issue with ModelPlot when splitting 3D and 4D SFS.

1.2.19 1.0.7

- Bug fixes and haplotype parsing in moments.LD.Parsing. (Issues #38 through #42, with thanks to Nathaniel Pope).

1.2.20 1.0.6

- Updates to installation, so that `pip` installs dependencies automatically.
- Protect against importing `matplotlib` if not installed.
- `Triallele` and `TwoLocus` now ensure using CSC format sparse matrix to avoid sparse efficiency warnings.
- Streamline test suite, which now works with `pytest`, as `python -m pytests tests`.

1.2.21 1.0.5

- Fixes install issues using pip: `pip install .` or `pip install git+https://bitbucket.org/simongravel/moments.git` is now functional.

1.2.22 1.0.4

- Stable importing of scipy.optimize nnls function.
- Fixes a plotting bug when ax was set to None (from @noscode - thanks!).

1.2.23 1.0.3

- Options in plotting scripts for showing and saving output.
- Add confidence interval computation for LD.
- Add parsing script for ANGSD frequency spectrum output.

Note that we started tracking changes between versions with version 1.0.2.

CHAPTER

TWO

INSTALLATION

`moments` now supports Python 3. Because Python is soon discontinuing support for Python 2, we do not actively ensure that moments remains fully compatable with Python 2, and strongly recommend using Python 3.

2.1 Using conda

`moments` is available via Bioconda.

The most recent release of `moments` can be installed by running

```
conda install -c bioconda moments
```

The conda channels must be set up to include bioconda, which can be done by running

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

2.2 Using pip

A simple way to install `moments` is via `pip`. `numpy`, `mpmath`, and `cython` are install requirements, but installing `moments` directly from the git repository using `pip` should install these dependencies automatically:

```
pip install git+https://bitbucket.org/simongravel/moments.git
```

This approach can also be used to install the development branch of `moments`:

```
pip install git+https://bitbucket.org/simongravel/moments.git@devel
```

Alternatively, you can clone the git repository

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

and then from within the moments directory (`cd moments`), run

```
pip install -r requirements.txt
pip install .
```

2.3 Dependencies and details

`moments` and `moments.LD` requires a number of dependencies. Minimally, these include

- numpy
- scipy
- cython
- mpmath
- demes

All dependencies are listed in *requirements.txt*, and can be install together using

```
pip install -r requirements.txt
```

We also strongly recommend installing `ipython` for interactive analyses.

If you are using conda, all dependencies can be installed by navigating to the moments directory and then running

```
conda install --file requirements.txt
```

Once dependencies are installed, to install `moments`, run the following commands in the moments directory:

```
python setup.py build_ext --inplace
python setup.py install
```

or

```
pip install .
```

Note that you might need sudo privileges to install in this way.

You should then be able to import `moments` in your python scripts. Entering an ipython or python session, type `import moments`. If, for any reason, you have trouble installing `moments` after following these steps, please submit an Issue.

If you use `Parsing` from `moments.LD`, which reads VCF-formatted files and computes LD statistics to compare to predictions from `moments.LD`, you will need to additionally install

- hdf5
- scikit-allel

CHAPTER

THREE

THE SITE FREQUENCY SPECTRUM

This page describes the Site Frequency Spectrum (SFS), how to compute its expectation using `moments`, manipulate spectra, implement demographic models using the `moments` API, and computing and saving spectra from a VCF.

If you use the SFS methods in `moments` in your research, please cite

- [Jouganous2017] Jouganous, J., Long, W., Ragsdale, A. P., & Gravel, S. (2017). Inferring the joint demographic history of multiple populations: beyond the diffusion approximation. *Genetics*, 206(3), 1549-1567.

3.1 The SFS

A site-frequency spectrum is a p-dimensional histogram, where p is the number of populations for which we have data. Thus, the shape of the SFS is $(n_0 + 1) \times (n_1 + 1) \times \ldots (n_{p-1} + 1)$, where n_i is the haploid sample size in population i. An entry of the SFS (call it `fs`) stores the number, density, or probability for SNP frequencies given by the index of that entry. That is, `fs[j, k, l]` is the number (or density) of mutations with allele frequencies `j` in population 0, `k` in population 1, and `l` in population 2. (Note that all indexing, as is typical in Python, is zero-based.)

3.1.1 Examples

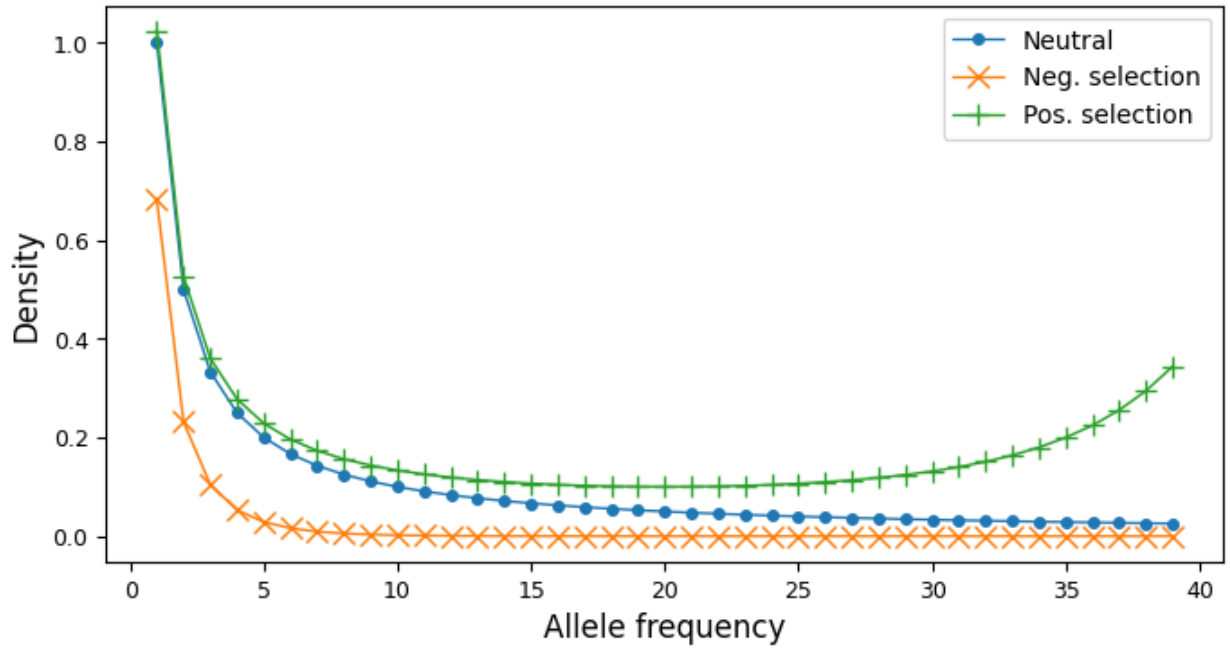
It can be helpful to visualize site-frequency spectra if you are new to working with them. In the single-population case, a SFS is a one-dimensional array. For variable biallelic loci and steady-state demography (no historical size changes, migrants, etc), the SFS is proportional to $1/i$, with total size depending on the mutation rate and sequence length. Historical size changes and demographic events perturb the SFS from this shape, as does negative or positive selection, skewing the SFS to lower or higher frequencies, resp.

```
import moments
import numpy as np
import matplotlib.pylab as plt

sample_size = 40

# A neutral SFS
fs_neu = moments.Demographics1D.snm([sample_size])
# SFS under negative selection
fs_neg = moments.Spectrum(
    moments.LinearSystem_1D.steady_state_1D(sample_size, gamma=-10)
)
# SFS under positive selection
fs_pos = moments.Spectrum(
    moments.LinearSystem_1D.steady_state_1D(sample_size, gamma=10)
```

(continues on next page)

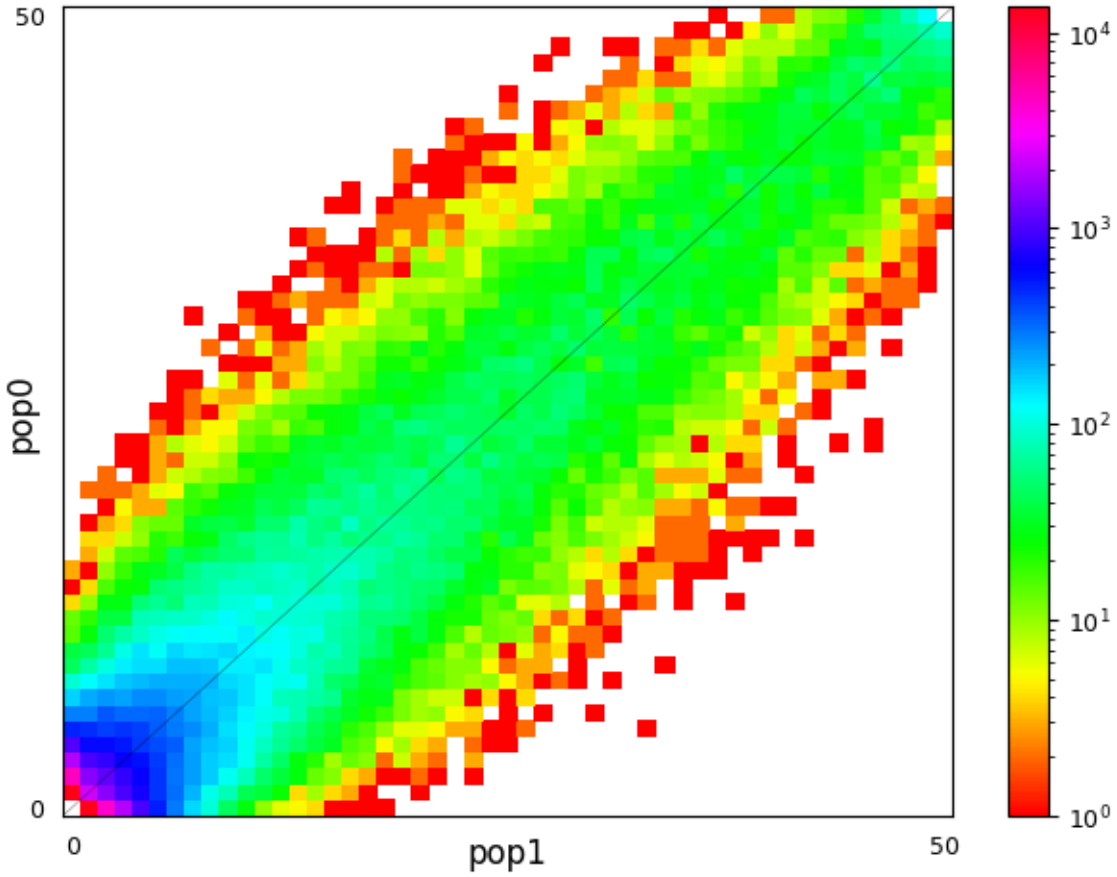
(continued from previous page)

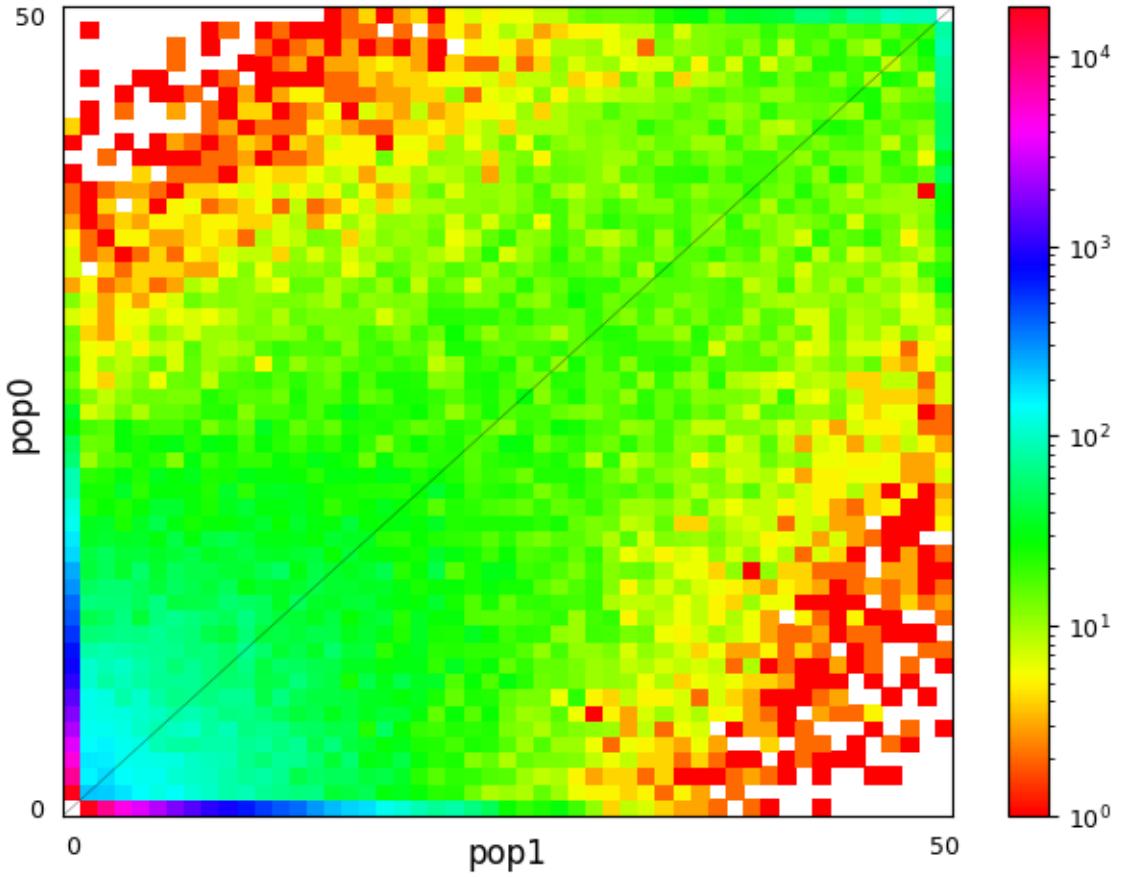
```
)

fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(fs_neu, ".-", ms=8, lw=1, label="Neutral")
ax.plot(fs_neg, "x-", ms=8, lw=1, label="Neg. selection")
ax.plot(fs_pos, "+-", ms=8, lw=1, label="Pos. selection")
ax.set_xlabel("Allele frequency")
ax.set_ylabel("Density")
ax.legend();
```



Multi-population SFS can be illustrated as multi-dimensional histograms, such as 2D heat maps. Here, we consider a very simple model of a population split and both derived populations are the same size as the ancestral population and do not exchange migrants. Allele frequencies in populations that split more recently will still be quite similar, while more distantly related populations are expected to have larger allele frequency differences.

```
sample_sizes = [50, 50]

# parameters of `split_mig` are (nu0, nu1, T, m)
# T is measured in units of 2Ne generations
fs_recent = moments.Demographics2D.split_mig((1, 1, 0.02, 0), sample_sizes)
fs_older = moments.Demographics2D.split_mig((1, 1, 0.15, 0), sample_sizes)

# assume theta = 20000, and then resample to fake data
fs_recent = (20000 * fs_recent).sample()
fs_older = (20000 * fs_older).sample()

moments.Plotting.plot_single_2d_sfs(fs_recent)
moments.Plotting.plot_single_2d_sfs(fs_older)
```


50
0
pop0
0
pop1
50
10^4
10^3
10^2
10^1
10^0



3.2 Spectrum objects in `moments`

SFS are stored as `moments.Spectrum` objects. If you are familiar with dadi's Spectrum objects, then you already will know your way around a `moments.Spectrum` object. `moments` has built off the `dadi` SFS construction, manipulation, and demographic specification, with minor adjustments that reflect the differences between the simulation engines and parameterizations.

`Spectrum` objects are a subclass of `numpy.masked_array`, so that standard array manipulation is possible. Indexing also works the same way as a typical array, so that `fs[2, 3, 5]` will return the entry in the SFS corresponding to allele frequencies $(2, 3, 5)$ (here, in a three-population SFS). Similarly, we can check if the SFS is masked at a given entry. For example, `fs.mask[0, 0]` returns whether the "fixed" bin (where no samples carry the derived allele) is ignored.

A `Spectrum` object has a few additional useful attributes:

- `fs.pop_ids`: A list of population IDs (as strings) for each population in the SFS.
- `fs.sample_sizes`: A list of sample sizes (as integers) corresponding to the shape of the SFS.
- `fs.folded`: If True, the SFS is folded, meaning we polarize allele frequencies by the minor allele frequency. If False, the SFS is polarized by the derived allele.

3.3 Manipulating SFS

Along with standard array manipulations, there are operations specific to SFS. Some of these are equivalent to standard array operations, but we ensure that the masking and population IDs are updated properly.

3.3.1 Folding

Folding a SFS removes information about how SNPs are polarized, so that the Spectrum stores counts of mutations with a given minor allele frequency. To fold a SFS, we call `fold()`, which returns a folded Spectrum object.

For example, the standard neutral model of sample size 10,

```
fs = moments.Demographics1D.snm([10])
fs
```

```
Spectrum([-- 1.0 0.4999999999999999 0.33333333333333326 0.25 0.2
 0.16666666666666666 0.14285714285714285 0.125 0.1111111111111111 --], folded=False, pop_
↪ids=None)
```

can be folded to the minor allele frequency, which updates the allele counts in the minor allele frequency bins and the mask:

```
fs_folded = fs.fold()
fs_folded
```

```
Spectrum([-- 1.1111111111111112 0.6249999999999999 0.4761904761904761
 0.41666666666666663 0.2 -- -- -- -- --], folded=True, pop_ids=None)
```

When folding multi-dimensional SFS, note that the folding occurs over the global minor allele frequency.

3.3.2 Projecting

SFS projection takes a Spectrum of some sample size and reduces the sample size in one or more populations. The output Spectrum sums over all possible down-samplings so that it is equivalent to having sampled a smaller sample size to begin with.

```
fs_proj = fs.project([6])
fs_proj
```

```
Spectrum([-- 0.9999999999999996 0.49999999999999994 0.33333333333333354
 0.24999999999999994 0.19999999999999996 --], folded=False, pop_ids=None)
```

For multi-dimensional frequency spectra, we must pass a list of sample sizes of equal length to the dimension of the SFS:

```
fs = moments.Spectrum(np.random.rand(121).reshape((11, 11)))
fs_proj = fs.project([6, 4])
fs_proj
```

```
Spectrum([[-- 1.9948878267740595 1.7283833831873054 1.73732504778059
  1.361451854778534]
 [1.9644248523024241 1.8493684567132138 1.9453737771234196
  1.7393877889305676 1.7118050950087182]
 [1.4658562378689899 1.9668516364659765 2.031982004524399
  1.9279764660450593 1.9350020193867488]
 [1.2960817961021882 1.8186376885188749 1.7464888972268522
  1.7926566537918327 2.0651198235724806]
 [1.2332051720644557 1.7444577194428192 1.712738339591321
  1.6715860671955958 2.3590054151219237]
 [1.676783201525725 1.5862885922524417 1.6987939560322936
  1.5759286146477345 1.8198224789028643]
 [2.0396880200564897 1.8901083442756392 1.8384809973409861
  1.84009209059203 --]], folded=False, pop_ids=None)
```

3.3.3 Marginalizing

If a population goes extinct, or if we want to subset a SFS to some focal populations, we use the `marginalize( )` function. This function takes a list of population indexes as input, and removes those indexes from the output SFS. The array operation is simply a sum over those axes, but the marginalization function also preserves population IDs if given.

For example, given a three-population spectrum

```
fs = moments.Spectrum(np.ones((5, 5, 5)), pop_ids=["A", "B", "C"])
fs
```

```
Spectrum([[[-- 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]]

 [[1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]]

 [[1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]]

 [[1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]]

 [[1.0 1.0 1.0 1.0 1.0]
```

(continues on next page)

(continued from previous page)

```
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 1.0]
  [1.0 1.0 1.0 1.0 --]]], folded=False, pop_ids=['A', 'B', 'C'])
```

we can view the one-population SFS, here the first population:

```
fs_marg = fs.marginalize([1, 2])
fs_marg
```

```
Spectrum([-- 25.0 25.0 25.0 --], folded=False, pop_ids=['A'])
```

or the joint two-population SFS for population indexes 1 and 2:

```
fs_marg = fs.marginalize([0])
fs_marg
```

```
Spectrum([[-- 5.0 5.0 5.0 5.0]
 [5.0 5.0 5.0 5.0 5.0]
 [5.0 5.0 5.0 5.0 5.0]
 [5.0 5.0 5.0 5.0 5.0]
 [5.0 5.0 5.0 5.0 --]], folded=False, pop_ids=['B', 'C'])
```

Note that the population IDs stay consistent after marginalizing.

3.3.4 Resampling

We can resample a new SFS from a given Spectrum using two approaches. First, a standard assumption is that entries in an "expected" SFS give the expectation of counts within each bin, and data follows a Poisson distribution with rates equal to the bin values. Then `sample()` creates a Poisson-sampled SFS:

```
fs = moments.Demographics1D.snm([10]) * 1000
fs_pois = fs.sample()
fs_pois
```

```
Spectrum([-- 1056 456 339 255 196 148 154 121 108 --], folded=False, pop_ids=None)
```

Alternatively, we could resample and enforce that we obtain a SFS with the same number of segregating sites:

```
fs_fixed = fs.fixed_size_sample(np.rint(fs.S()))
print(f"number of sites in input:", f"{fs.S():.2f}")
print(f"number of sites in resampled SFS:", fs_fixed.S())
fs_fixed
```

```
number of sites in input: 2828.97
number of sites in resampled SFS: 2829
```

```
Spectrum([-- 977 507 312 275 201 172 140 138 107 --], folded=False, pop_ids=None)
```

3.4 Demographic events

When defining demographic models with multiple populations, we need to apply demographic events such as population splits, mergers, and admixtures. These operations often change the dimension or size of the SFS, so they do not act in-place. Instead, they return a new Spectrum object, similar to the manipulations in the previous section.

3.4.1 Population splits and branches

New in `moments` version 1.1, the Spectrum class includes functions to directly apply demographic events. A population split is called using `fs.split(idx, n0, n1)`, where the population indexed by `idx` splits into `n0` and `n1` lineages. The `split` function also takes a `new_ids` keyword argument, where we can specify the population IDs of the two new populations after the split. Note that `n0` and `n1` cannot sum to larger than the current sample size of the population that we are splitting.

For example, to split a single population with 6 tracked lineages into two populations with 3 lineages in each population:

```
fs = moments.Demographics1D.snm([6])
fs_split = fs.split(0, 3, 3)
fs_split
```

```
Spectrum([[-- 0.4999999999999997 0.09999999999999999 0.016666666666666653]
 [0.4999999999999997 0.2999999999999993 0.14999999999999986
  0.049999999999999996]
 [0.09999999999999999 0.14999999999999986 0.14999999999999997
  0.09999999999999995]
 [0.016666666666666653 0.049999999999999996 0.09999999999999995 --]], folded=False, pop_
↪ids=None)
```

If we use `new_ids`, we can also keep track of population ids after a split event:

```
fs = moments.Demographics2D.snm([6, 2], pop_ids=["A", "B"])
fs
```

```
Spectrum([[-- 0.24999999999999994 0.017857142857142853]
 [0.7500000000000001 0.21428571428571433 0.03571428571428571]
 [0.2678571428571428 0.17857142857142852 0.053571428571428506]
 [0.11904761904761908 0.1428571428571428 0.07142857142857138]
 [0.053571428571428506 0.10714285714285701 0.08928571428571426]
 [0.021428571428571408 0.07142857142857144 0.10714285714285715]
 [0.00595238095238095 0.035714285714285705 --]], folded=False, pop_ids=['A', 'B'])
```

```
fs_split = fs.split(0, 4, 2, new_ids=["C", "D"])
fs_split
```

```
Spectrum([[[-- 0.24999999999999994 0.017857142857142853]
  [0.24999999999999994 0.07142857142857141 0.011904761904761904]
  [0.017857142857142853 0.011904761904761899 0.0035714285714285674]]

 [[0.49999999999999994 0.14285714285714285 0.023809523809523808]
  [0.14285714285714285 0.09523809523809523 0.028571428571428543]
  [0.023809523809523808 0.028571428571428543 0.014285714285714268]]
```

(continues on next page)

(continued from previous page)

```
[[0.10714285714285711 0.0714285714285714 0.0214285714285714]
 [0.0714285714285714 0.0857142857142856 0.0428571428571428]
 [0.0214285714285714 0.042857142857142795 0.0357142857142857]]

[[0.0238095238095238 0.028571428571428543 0.01428571428571427]
 [0.02857142857142854 0.057142857142857086 0.047619047619047616]
 [0.014285714285714268 0.047619047619047616 0.07142857142857142]]

[[0.003571428571428567 0.007142857142857133 0.00595238095238095]
 [0.007142857142857134 0.023809523809523805 0.035714285714285705]
 [0.00595238095238095 0.035714285714285705 --]]], folded=False, pop_ids=['C', 'B', 'D'])
```

As of version 1.1.5, we can apply a "branch" event. This is conceptually similar to a split, but simpler in that a child population branches off from a parental population. In this case, we just need to give the sample size of the new child population (and it's new population ID), and the parental population is left with the same number of lineages minus the size of the new population, and its population ID (if given) remains unchanged.

```
fs = moments.Demographics1D.snm([5], pop_ids=["A"])
fs_branch = fs.branch(0, 2, new_id="B")
fs_branch
```

```
Spectrum([[-- 0.40000000000000013 0.05000000000000001]
 [0.6000000000000001 0.30000000000000004 0.1]
 [0.15000000000000002 0.2 0.15000000000000002]
 [0.03333333333333334 0.10000000000000003 --]], folded=False, pop_ids=['A', 'B'])
```

Note: Previous versions of `moments` required calling functions such as `moments.Manips.split_1D_to_2D(fs, n0, n1)` or `moments.Manips.split_3D_to_4D_2(fs, n0, n1)`. The new API (`fs.split(idx, n0, n1)`) wraps the different split functions in `moments.Manips` so that we don't need to worry about picking the correct split function.

3.4.2 Admixture and mergers

Here, we consider two types of admixture events. First, two populations mix with given proportions to form a new population (which we will call an "admix" event). And second, one population contributes some proportion to another population in the SFS (which we call a "pulse migration" event). In both cases, lineages within the SFS are moved from one or more populations to another, and its size and possibly dimension can change.

To mix two population with a given proportion, we use `fs.admix(idx0, idx1, num_lineages, proportion)`, where `proportion` is the proportion of the new population that comes from population `idx0`, and 1-proportion comes from population indexed by `idx1`. The number of lineages is the sample size in the new admixed population, and the sample sizes in the source populations necessarily decrease by that same amount. Note that if the sample size of a source population equals the number of lineages that are moved, that source population no longer exists and the dimension decreases by one.

For example, in a two-population SFS, we can look at a few different scenarios of admixture and sample sizes:

```
fs = moments.Spectrum(np.ones((11, 11)))
print("original SFS has sample size", fs.sample_sizes)
```

(continues on next page)

(continued from previous page)

```
fs_admix = fs.admix(0, 1, 10, 0.25)
print("admix SFS has size", fs_admix.sample_sizes, "after moving 10 lineages")
fs_admix2 = fs.admix(0, 1, 5, 0.5)
print("second admix SFS has size", fs_admix2.sample_sizes, "after moving 5 lineages")
```

```
original SFS has sample size [10 10]
admix SFS has size [10] after moving 10 lineages
second admix SFS has size [5 5 5] after moving 5 lineages
```

And to account for population IDs after admixture:

```
fs = moments.Spectrum(np.ones((9, 7)), pop_ids=["A", "B"])
print("original SFS has size", fs.sample_sizes, "and pop ids", fs.pop_ids)
fs_admix = fs.admix(0, 1, 4, 0.25, new_id="C")
print("admix SFS has size", fs_admix.sample_sizes, "and pop ids", fs_admix.pop_ids,
    "after moving 4 lineages into new population C")
```

```
original SFS has size [8 6] and pop ids ['A', 'B']
admix SFS has size [4 2 4] and pop ids ['A', 'B', 'C'] after moving 4 lineages into new␣
↪population C
```

3.5 Integration

`moments` integrates the SFS forward in time by calling `fs.integrate( )`. At a minimum, we need to pass the population size(s) `nu` and the integration time `T`. All parameters are scaled by a reference effective population size, so that time is measured in units of $2N_e$ generations, sizes are relative to this same N_e, and mutation and migration rates and the selection coefficient is scaled by $2N_e$.

3.5.1 Size functions

The `integrate( )` function can take either a list of relative sizes, equal to the number of populations represented by the SFS, or it can take a function that returns a list of population sizes over time.

For example, to integrate a two-population SFS with the first population having relative size 2.0 (double the reference size), and the second having size 0.1 (one-tenth the relative size) for 0.05 time units:

```
fs = moments.Demographics2D.snm([10, 10])
fs.integrate([2.0, 0.1], 0.05)
```

To specify a size function that changes over time, for example an exponential growth model, we can instead pass a size function to the integration method:

```
fs = moments.Demographics1D.snm([10])
nu0 = 0.5
nuF = 2.0
T = 0.2
nu_func = lambda t: [nu0 * np.exp(np.log(nuF / nu0) * t / T)]
print("size at start of epoch:", nu_func(0))
print("size at end of epoch:", nu_func(T))
fs.integrate(nu_func, T)
```

```
size at start of epoch: [0.5]
size at end of epoch: [1.9999999999999996]
```

3.5.2 Integration time and time units

Unlike coalescent simulators, such as `msprime`, integration times in `moments` are in units of $2N_e$ generations. Thus, typical integration times for many demographic scenarios could be much smaller than one.

Times are not cummulative when integrating multiple epochs - each time `integrate( )` is called, internally time starts from zero by default. Thus, when defining multiple epochs with size functions, keep in mind that time for that epoch runs from zero to the integration time `T`.

3.5.3 Migration rates

Migration between populations is specified by the migration matrix, with has shape p $timesp$, where p is the number of populations represented by the SFS. The i-th row of the migration matrix gives the migration rates from each other population *into* the population indexed by i. Because rates are rescaled by the effective population size, the entry `M[i, j]` gives the migration rate `2*Ne*m_ij`, where `m_ij` is the per-generation probability of a lineage in population `i` having its parent in population `j`. Note that the diagonal elements of `M` are ignored.

For example, to integrate a two-population SFS with migration:

```
fs = moments.Demographics2D.snm([10, 10])
M = np.array([
    [0, 2.0],
    [0.75, 0]
])
fs.integrate([2, 3], 0.05, m=M)
```

3.5.4 Mutation rates and mutation model

By default, `moments` uses an infinite-sites model (ISM). Then the mutation rate θ is the population-size scaled mutation rate multiplied by the number of loci: `theta = 4*Ne*u*L`. By default, `theta` is set to 1.

Luckily, we do not often need to worry about setting `theta`, because the ISM guarantees that the expected count in each frequency bin of the SFS scales linearly in the mutation rate. This means that we can happily integrate with the default `theta` and only rescale the SFS at the end:

```
theta = 100
fs_theta = moments.LinearSystem_1D.steady_state_1D(20) * 100
fs_theta = moments.Spectrum(fs_theta)
fs_theta.integrate([2.0], 0.1, theta=theta)

fs = moments.Demographics1D.two_epoch((2.0, 0.1), [20]) # default theta = 1
fs = theta * fs

print(fs_theta.S())
print(fs.S())
```

```
395.6948077081298
395.69480770813
```

Reversible mutations

Unlike `dadi`, which solves the diffusion equation directly and can only simulate under the ISM, the moments-based engine in `moments` lets us accurately track the density of the "fixed" bins. That is, we can compute not just the distribution of segregating mutation frequencies, but also the probability that a locus is monomorphic in a sample for the derived or ancestral allele.

To compute a SFS in which we track monomorphic loci, we use a reversible mutation model, which we specify by setting `finite_genome=True`. When simulating under the finite genome model, the mutation rate is no longer scaled by the number of loci, `L`. Instead, the mutation rates are simply `theta_fd=4*Ne*u` and `theta_bd=4*Ne*v` where `u` and `v` are the forward and backward mutation rates, respectively. Therefore, `theta_fd` and `theta_bd` are typically much less than 1 (and in fact the model breaks down for scaled mutation rates around 1).

To simulate under the reversible mutation model, we first initialize the steady-state SFS with `mask_corners=False`, and then apply demographic events as normal and integrate using `finite_genome=True`:

```
theta_fd = 0.0005 # 4*Ne*u, with Ne = 1e4 and u = 1.25e-8
theta_bd = 0.001 # the backward mutation rate is double the forward rate
fs = moments.LinearSystem_1D.steady_state_1D_reversible(
    20, theta_fd=theta_fd, theta_bd=theta_bd) # sample size = 20
fs = moments.Spectrum(fs, mask_corners=False)

fs.integrate(
    [5.0], 0.2, finite_genome=True, theta_fd=theta_fd, theta_bd=theta_bd)
```

Note that if the forward and backward mutation rates are equal, we can use `theta` to set both mutation rates (which must be set, as `theta` must be less than 1).

Illustration: ancestral state misidentification

In SFS analyses, a typical confounder is the misidentification of the ancestral allele. This occurs because polarization requires estimating the ancestral state of a locus, which is typically done by comparing to one or more outgroup species in a sequence alignment. For humans, we typically use chimpanzee and other great apes to infer the ancstral allele.

At longer evolutionary timescales, it is not uncommon for multiple independent mutations to occur at the same locus, so that when comparing to an outgroup species we classify some derived mutations as ancestral and some ancestral mutations as derived. For humans, the rate of ancestral misidentification is typically in the 1-3% range, depending on the method used to polarize alleles.

For example, we can simulate using rough parameters ($u = 1.25 \times 10^{-8}$, $N_e = 10^4$, divergence of 6 million years, and a generation time of 25 years) and symmetric mutation rates to see the effect of polarizing based on the allele in a chimp sequence. Here, if the chimp carries the derived allele, we will instead assume the ancestral allele is derived:

```
Ne = 1e4
u = 1.25e-8
theta = 4 * Ne * u
generation_time = 25
divergence_years = 6e6
T = divergence_years / generation_time / 2 / Ne

fs = moments.LinearSystem_1D.steady_state_1D_reversible(
    101, theta_fd=theta, theta_bd=theta)
fs = moments.Spectrum(fs, mask_corners=False)

fs = fs.split(0, 100, 1)
```

(continues on next page)

(continued from previous page)

```
fs.integrate([1, 1], T, finite_genome=True, theta=theta)

fs_polarized = fs[:,0] + fs[::-1,1]
fs_polarized.mask_corners()
```

Then visualizing using `moments.Plotting.plot_1d_fs(fs_polarized)`, we can see the uptick at high-frequency variants due to ancestral misidentification - that is, recurrent mutations along the lineage leading from humans to chimps:

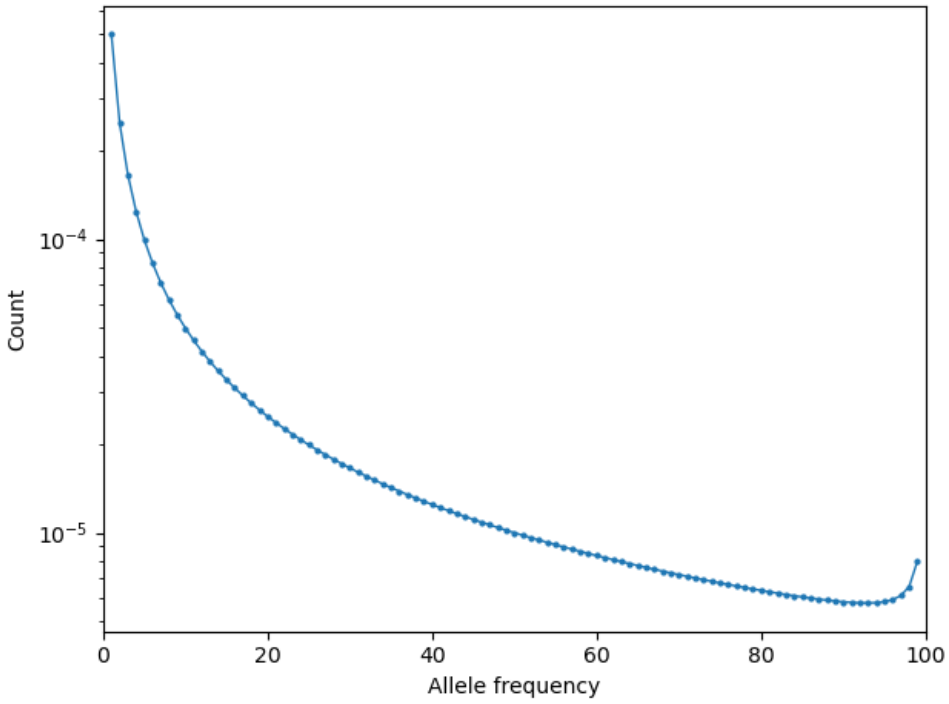


Fig. 3.1: Excess of high-frequency derived mutations due to ancestral misspecification.

3.5.5 Selection and dominance

One of the great benefits to forward simulators is their ability to include the effects of selection and dominance with little extra cost. In the selection model implemented in `moments`, genotype fitnesses are given relative to the ancestral homozygous genotype (i.e. relative fitness of *aa* is 1), so that heterozygous genotypes (*Aa*) have relative fitness $1+2hs$ and homozygous derived genotypes (*AA*) have relative fitness $1+2s$.

When $h = 1/2$, selection is additive (or genic), which corresponds to haploid copies of the derived allele having average fitness $1 + s$. If `h` is unspecified, the selection model defaults to additivity ($h = 1/2$), and if `gamma` is unspecified, we default to neutrality.

Note: We assume $|s| \ll 1$, so that s^2 and higher order terms can be ignored. For strong selection in a moments framework, see recent advances from [Krukov2021].

`moments` takes scaled selection coefficients $\gamma = 2N_e s$ and dominance coefficients h as keyword parameters when initializing the SFS and integrating. The reference N_e is often taken as the ancestral effective population size.

```
gamma = -5
h = 0.1
ns = 30

fs = moments.LinearSystem_1D.steady_state_1D(ns, gamma=gamma, h=h)
fs = moments.Spectrum(fs)
print("Tajima's D (before expansion):", fs.Tajima_D())

fs.integrate([3], 0.2, gamma=gamma, h=h)
print("Tajima's D (after expansion):", fs.Tajima_D())
```

```
Tajima's D (before expansion): -0.643870774090141
Tajima's D (after expansion): -1.1502872304492777
```

Simulating selection with multiple populations works similarly. We can specify `gamma` and `h` as scalar values, which implies that the allele has the same selection and dominance effect in each population. We can instead simulate population-specific selection and dominance coefficients by setting `gamma` and/or `h` as a list of length equal to the number of populations in the spectrum, with indexing matching the ordering of the populations in the spectrum object.

3.5.6 Ancient samples and frozen populations

So far, in all the examples we've seen the output SFS integrates all populations until the same end time. If one or more of the sampled populations are non-contemporary, we need to "freeze" those populations at their time of sampling. This is done by specifying which populations to freeze using the `frozen` argument.

For example, if we sample two populations that split 100kya, and one population consisting of ancient samples from 20kya, we integrate the first 80 thousand years as normal, and then the last 20 thousand years with the ancient population frozen:

```
Ne = 1e4
generation_time = 25
T1 = 80e3 / 2 / Ne / generation_time
T2 = 20e3 / 2 / Ne / generation_time
migrate = 0.5

fs = moments.Demographics2D.snm([10, 10])
fs.integrate([1, 1], T1, m=[[0, migrate], [migrate, 0]])
fs.integrate([1, 1], T1, m=[[0, migrate], [migrate, 0]], frozen=[False, True])
```

3.6 Computing summary statistics

`moments` allows us to compute a handful of summary statistics from the SFS. For single populations, we can get Watterson's θ, the diversity π, or Tajima's D directly from the SFS:

```
fs = moments.Demographics1D.two_epoch((3.0, 0.2), [20])
print("Watterson's theta:", fs.Watterson_theta())
print("Diversity:", fs.pi())
snm = moments.Demographics1D.snm([20])
print("Tajima's D at steady state:", snm.Tajima_D())
print("Tajima's D after expansion:", fs.Tajima_D())
```

```
Watterson's theta: 1.291270898392208
Diversity: 1.128986048415916
Tajima's D at steady state: 3.1116722926989843e-16
Tajima's D after expansion: -0.37656997453348207
```

For multi-population spectra, we can also compute FST using Weir and Cokerham's (1984) method, which generalizes to any number of populations greater than one:

```
fs = moments.Demographics2D.snm([10, 10])
print("FST immediately after split:", fs.Fst())
fs.integrate([1, 1], 0.05)
print("FST after isolation of 0.05*2*Ne gens:", fs.Fst())
fs.integrate([1, 1], 0.05)
print("FST after isolation of 0.1*2*Ne gens:", fs.Fst())
```

```
FST immediately after split: 0.05263157894736842
FST after isolation of 0.05*2*Ne gens: 0.09774436090225562
```

```
FST after isolation of 0.1*2*Ne gens: 0.13875598086124397
```

Note that FST is sensitive to sample sizes: smaller sample sizes artificially inflate the "true" divergence.

```
print("10 samples each:", moments.Demographics2D.snm([10, 10]).Fst())
print("100 samples each:", moments.Demographics2D.snm([100, 100]).Fst())
```

```
10 samples each: 0.05263157894736842
```

```
100 samples each: 0.005025125628140709
```

3.7 Compute SFS from VCF

`moments` supports computing a SFS from files in VCF format, given a population information file. This takes two steps. We first parse the VCF using and we then pass that data dictionary to the Spectrum class:

```
data_dict = moments.Misc.make_data_dict_vcf(vcf_filename, popinfo_filename)
fs = moments.Spectrum.from_data_dict(data_dict)
```

3.8 Plotting the SFS

`moments` comes pre-installed with a number of plotting functions, which can be called from `moments.Plotting`. These include functions to plot individual SFS, or to compare two SFS (for example, to compare a model to data). These functions can be used out-of-the-box, or serve as inspiration for your own `matplotlib` adventures. To see what plotting functions are available and view their documentation, head to the *moments API*.

3.9 References

CHAPTER

FOUR

SFS INFERENCE

4.1 Computing likelihoods

Following [Sawyer1992] the distribution of mutation frequencies is treated as a Poisson random field, so that composite likelihoods (in which we assume mutations are independent) are computed by taking Poisson likelihoods over bins in the SFS. We typically work with log-likelihoods, so that the log-likelihood of the data (D) given the model (M) is

$$\log \mathcal{L} = \sum_i D_i \log M_i - M_i - \log D_i!$$

where i indexes the bins of the SFS.

Likelihoods can be computed from `moments.Inference`:

```
import moments
import numpy as np

theta = 1000
model = theta * moments.Demographics1D.snm([10])

data = model.sample()

print(model)
print(data)
```

```
[-- 1000.0 499.9999999999999 333.33333333333326 250.0 200.0
 166.66666666666666 142.85714285714286 125.0 111.1111111111111 --]
[-- 956 529 334 246 205 164 146 123 127 --]
```

```
print(moments.Inference.ll(model, data))
```

```
-36.09050060346999
```

When simulating under some demographic model, we usually use the default `theta` of 1, because the SFS scales linearly in the mutation rate. When comparing to data in this case, we need to rescale the model SFS. It turns out that the maximum-likelihood rescaling is that which makes the total number of segregating sites in the model equal to the total number in the data:

```
data = moments.Spectrum([0, 3900, 1500, 1200, 750, 720, 600, 400, 0])
model = moments.Demographics1D.two_epoch((2.0, 0.1), [8])

```

(continues on next page)

(continued from previous page)

```
print("Number of segregating sites in data:", data.S())
print("Number of segregating sites in model:", model.S())
print("Ratio of segregating sites:", data.S() / model.S())

opt_theta = moments.Inference.optimal_sfs_scaling(model, data)
print("Optimal theta:", opt_theta)
```

```
Number of segregating sites in data: 9070.0
Number of segregating sites in model: 2.7771726368386327
Ratio of segregating sites: 3265.911481226729
Optimal theta: 3265.911481226729
```

Then we can compute the log-likelihood of the rescaled model with the data, which will give us the same answer as `moments.Inference.ll_multinom` using the unscaled data:

```
print(moments.Inference.ll(opt_theta * model, data))
print(moments.Inference.ll_multinom(model, data))
```

```
-59.880644681554486
-59.880644681554486
```

4.2 Optimization

`moments` optimization is effectively a wrapper for `scipy` optimization routines, with some features specific to working with SFS data. In short, given a demographic model defined by a set of parameters, we try to find those parameters that minimize the negative log-likelihood of the data given the model. There are a number of optimization functions available in `moments.Inference`:

- `optimize` and `optimize_log`: Uses the BFGS algorithm.
- `optimize_lbfgsb` and `optimize_log_lbfgsb`: Uses the L-BFGS-B algorithm.
- `optimize_log_fmin`: Uses the downhill simplex algorithm on the log of the parameters.
- `optimize_powell` and `optimize_log_powell`: Uses the modified Powell's method, which optimizes slices of parameter space sequentially.

More information about optimization algorithms can be found in the scipy documentation.

With each method, we require at least three inputs: 1) the initial guess, 2) the data SFS, and 3) the model function that returns a SFS of the same size as the data.

Additionally, it is common to set the following:

- `lower_bound` and `upper_bound`: Constraints on the lower and upper bounds during optimization. These are given as lists of the same length of the parameters.
- `fixed_params`: A list of the same length of the parameters, with fixed values given matching the order of the input parameters. `None` is used to specify parameters that are still to be optimized.
- `verbose`: If an integer greater than 0, prints updates of the optimization procedure at intervals given by that spacing.

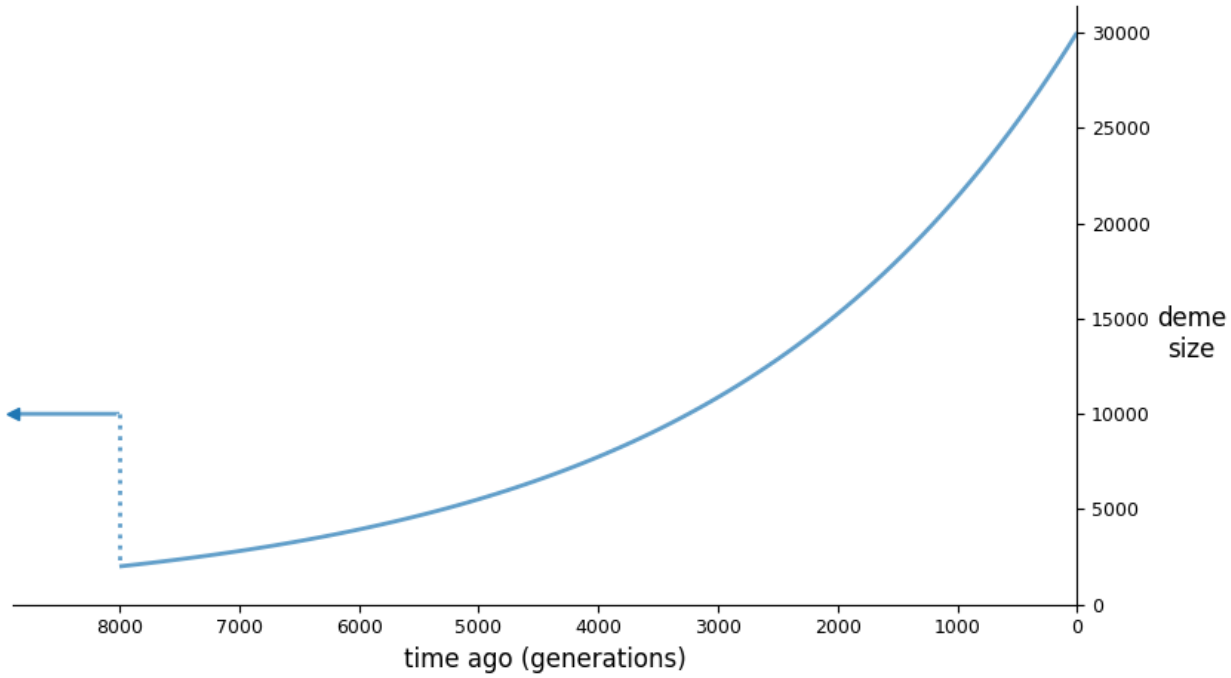
For a full description of the various inference functions, please see the *SFS inference API*.

4.2.1 Single population example

As a toy example, we'll generate some fake data from a demographic model and then reinfer the input parameters of that demographic model. The model is an instantaneous bottleneck followed by exponential growth, implemented in `moments.Demographics1D.bottlegrowth`, which takes parameters `[nuB, nuF, T]` and the sample size. Here `nuB` is the bottleneck size (relative to the ancestral size), `nuF` is the relative final size, and `T` is the time in the past the bottleneck occurred (in units of $2N_e$ generations).

```
nuB = 0.2
nuF = 3.0
T = 0.4

n = 60  # the haploid sample size

fs = moments.Demographics1D.bottlegrowth([nuB, nuF, T], [n])

theta = 2000  # the scaled mutation rate (4*Ne*u*L)
fs = theta * fs
data = fs.sample()
```

The input demographic model (assuming an N_e of 10,000), plotted using demesdraw:



We then set up the optimization inputs, including the initial parameter guesses, lower bounds, and upper bounds, and then run optimization. Here, I've decided to use the log-L-BFGS-B method, though there are a number of built in options (see previous section).

```
p0 = [0.2, 3.0, 0.4]
lower_bound = [0, 0, 0]
upper_bound = [None, None, None]
p_guess = moments.Misc.perturb_params(p0, fold=1,
    lower_bound=lower_bound, upper_bound=upper_bound)
```

(continues on next page)

(continued from previous page)

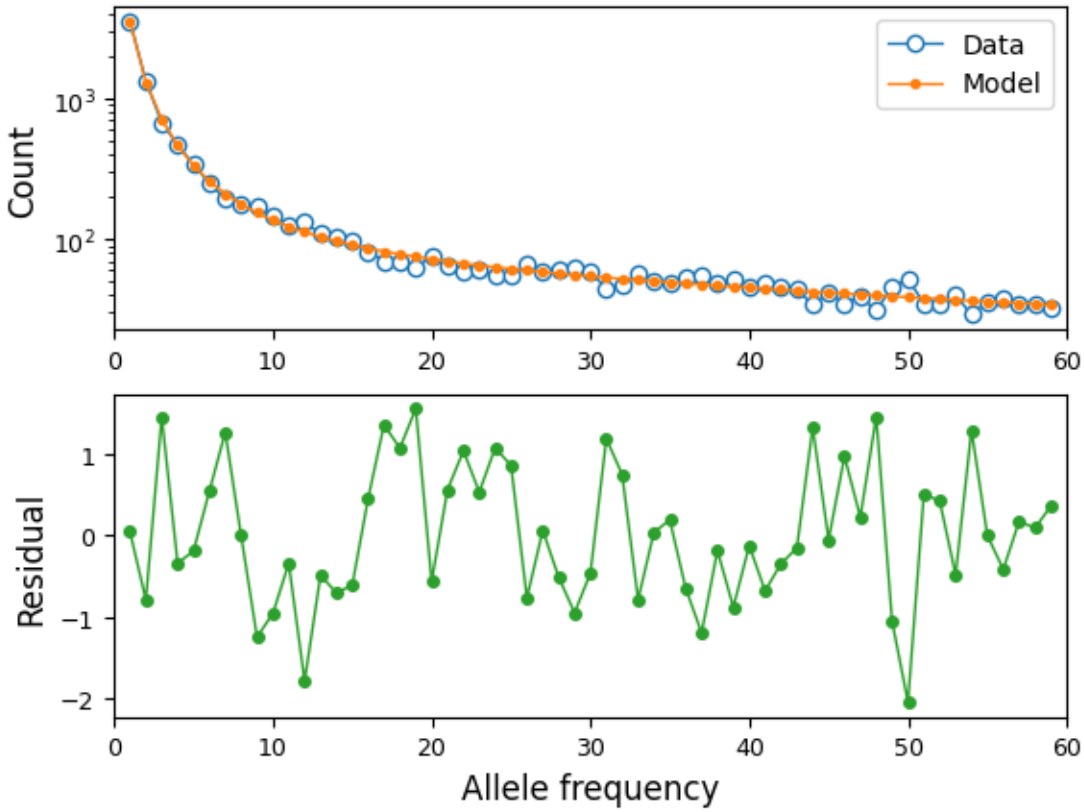
```
model_func = moments.Demographics1D.bottlegrowth

opt_params = moments.Inference.optimize_log_lbfgsb(
    p0, data, model_func,
    lower_bound=lower_bound,
    upper_bound=upper_bound)

model = model_func(opt_params, data.sample_sizes)
opt_theta = moments.Inference.optimal_sfs_scaling(model, data)
model = model * opt_theta
```

The reinferred parameters:

```
Params        nuB        nuF         T        theta
Input         0.2        3.0         0.4        2000
Refit         0.1874        2.977         0.403         2.074e+03
```

We can also visualize the fit of the model to the data:

```
moments.Plotting.plot_1d_comp_Poisson(model, data)
```



Confidence intervals

We're often interested in estimating the precision of the inferred parameters from our best fit model. To do this, we can compute a *confidence interval* for each free parameter from the model fit. Methods implemented in `moments` to compute, particularly the method based on the Godambe Information Matrix [Coffman2016], were first implemented in dadi by Alec Coffman, who's paper should be cited if these methods are used.

See the *API documentation for uncertainty functions* for information on their usage.

4.2.2 Two population example

Here, we will create some fake data for a two-population split-migration model, and then re-infer the input parameters to the model used to create that data. This example uses the `optimize_log_fmin` optimization function. We'll also use the `FIM_uncert` function to compute uncertainties (reported as standard errors).

```
input_theta = 10000
params = [2.0, 3.0, 0.2, 2.0]
model_func = moments.Demographics2D.split_mig
model = model_func(params, [20, 20])
model = input_theta * model
data = model.sample()

p_guess = [2, 2, .1, 4]
lower_bound = [1e-3, 1e-3, 1e-3, 1e-3]
upper_bound = [10, 10, 1, 10]

p_guess = moments.Misc.perturb_params(
    p_guess, lower_bound=lower_bound, upper_bound=upper_bound)

opt_params = moments.Inference.optimize_log_fmin(
    p_guess, data, model_func,
    lower_bound=lower_bound, upper_bound=upper_bound,
    verbose=20) # report every 20 iterations

refit_theta = moments.Inference.optimal_sfs_scaling(
    model_func(opt_params, data.sample_sizes), data)

uncerts = moments.Godambe.FIM_uncert(
    model_func, opt_params, data)

print_params = params + [input_theta]
print_opt = np.concatenate((opt_params, [refit_theta]))

print("Params\tnu1\tnu2\tT_div\tm_sym\ttheta")
print(f"Input\t" + "\t".join([str(p) for p in print_params]))
print(f"Refit\t" + "\t".join([f"{p:.4}" for p in print_opt]))
print(f"Std-err\t" + "\t".join([f"{u:.3}" for u in uncerts]))

moments.Plotting.plot_2d_comp_multinom(
    model_func(opt_params, data.sample_sizes), data)
```

```
60      , -1991.09    , array([ 3.32995    ,  2.08331    ,  0.184467   ,  3.04554    ])
```

```
80      , -1329.85    , array([ 1.5606     ,  2.83455    ,  0.237688   ,  2.94619    ])
```

```
100     , -1252.15    , array([ 1.78864    ,  2.51295    ,  0.235066   ,  2.54749    ])
```

```
120     , -1243.31    , array([ 1.85429    ,  2.44477    ,  0.214185   ,  2.54415    ])
```

```
140     , -1205.95    , array([ 1.90496    ,  3.03861    ,  0.189469   ,  2.16029    ])
```

```
160     , -1188.91    , array([ 2.09831    ,  2.95263    ,  0.201107   ,  1.93286    ])
```

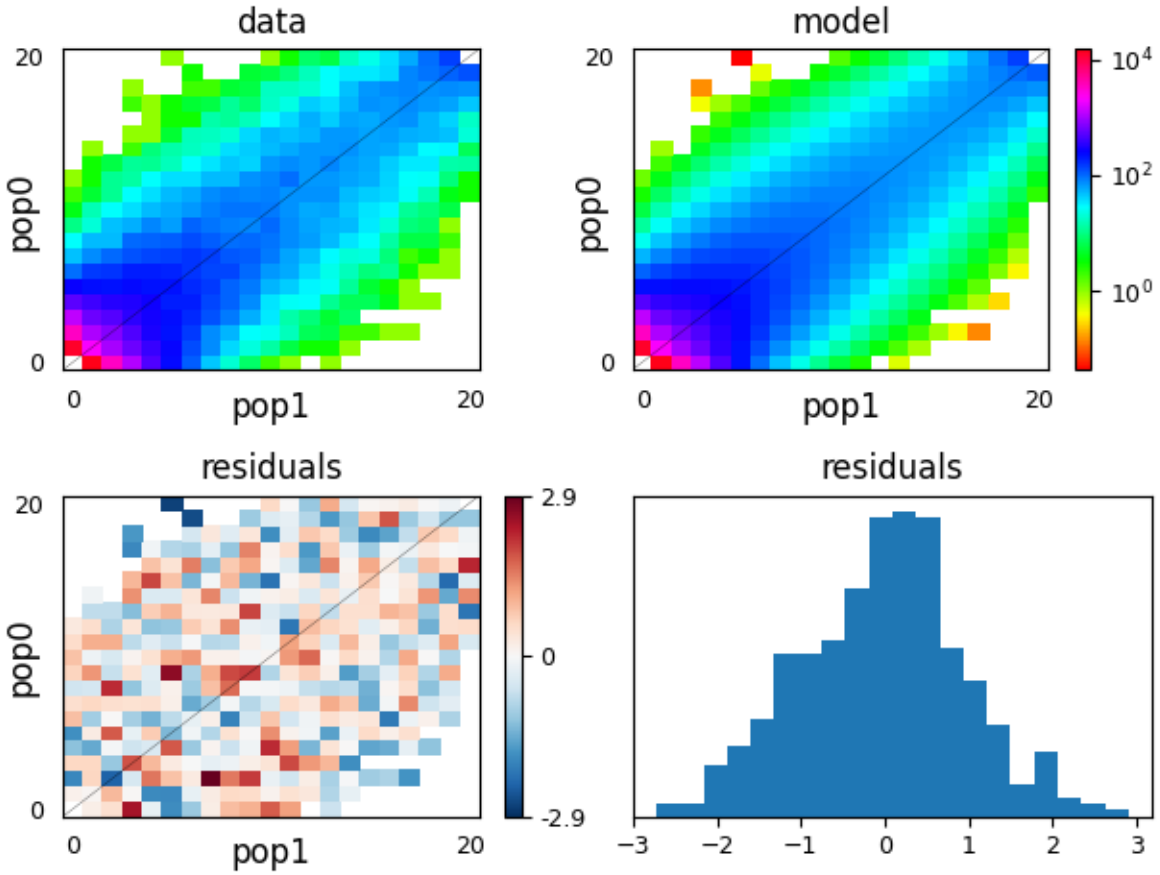
```
180     , -1188.21    , array([ 2.08727    ,  3.03551    ,  0.198968   ,  1.96291    ])
```

```
200     , -1187.61    , array([ 2.0709     ,  3.0153     ,  0.197898   ,  1.93797    ])
```

```
220     , -1187.6     , array([ 2.07395    ,  3.01909    ,  0.197498   ,  1.9306     ])
```

```
240     , -1187.6     , array([ 2.07377    ,  3.0196     ,  0.197521   ,  1.93134    ])
```

```
Params         nu1         nu2         T_div         m_sym         theta
Input          2.0         3.0           0.2           2.0         10000
Refit          2.074         3.019         0.1975          1.931         9.929e+03
Std-err          0.0414          0.0689          0.0044          0.0738          69.7
```



Above, we can see that we recovered the parameters used to simulate the data very closely, and we used `moments`'s plotting features to visually compare the data to the model fit.

4.3 References

CHAPTER

FIVE

MULTI-POPULATION LD STATISTICS

Using moment equations for the two-locus haplotype distribution, `moments.LD` lets us compute a large family of linkage disequilibrium statistics in models with arbitrary mutation and recombination rates and flexible demographic history with any number of populations. The statistics are stored in a different way that the SFS, but much of the API for implementing demographic events and integration is largely consistent between the SFS and LD methods.

If you use `moments.LD` in your research, please cite:

- [Ragsdale2019]: Ragsdale, A. P. & Gravel, S. (2019). Models of archaic admixture and recent history from two-locus statistics. *PLoS Genetics*, 15(6), e1008204.
- [Ragsdale2020]: Ragsdale, A. P. & Gravel, S. (2020). Unbiased estimation of linkage disequilibrium from unphased data. *Mol Biol Evol*, 37(3), 923-932.

5.1 Linkage disequilibrium

The LD statistics that `moments.LD` computes are low-order summaries of expected LD between pairs of loci. In particular, we compute $\mathbb{E}[D^2]$, the expectation of the numerator of the familiar r^2 measure of LD. From this system of equations, we also compute $\mathbb{E}[Dz] = \mathbb{E}[D(1-2p)(1-2q)]$, where p and q are the allele frequencies at the left and right loci, respectively; and we also compute $\pi_2 = \mathbb{E}[p(1-p)q(1-q)]$, a measure of the "joint heterozygosity" of the two loci [Hill1968].

These statistics are stored in a list of arrays, where each list element corresponds to a given recombination rate, $\rho = 4N_e r$, where r is the recombination probability separating loci. The length of the list is the length of the number of recombination rates given, plus one, as the last entry stores the single-locus expected heterozygosity:

```
import moments, moments.LD
theta = 0.001 # the mutation rate 4*Ne*u
rho = [0, 1, 10] # recombination rates 4*Ne*r between loci
y = moments.LD.Demographics1D.snm(rho=rho, theta=theta) # steady-state expectations
y
```

```
LDstats([[1.38888889e-07 1.11111111e-07 3.05555556e-07]
 [8.59375000e-08 6.25000000e-08 2.81250000e-07]
 [2.01612903e-08 8.06451613e-09 2.54032258e-07]], [0.001], num_pops=1, pop_ids=None)
```

Here, we can see the decay of LD with increasing recombination rate, and also that the heterozygosity equals the scaled mutation rate at steady-state, as expected. On any LD object, we can get the list of statistics present by calling:

```
y.names()
```

```
(['DD_0_0', 'Dz_0_0_0', 'pi2_0_0_0_0'], ['H_0_0'])
```

The underscores index the populations for that statistic, so `DD_0_0` represents $\mathbb{E}[D_0 D_0] = \mathbb{E}[D_0^2]$, `Dz_0_0_0` represents $\mathbb{E}[D_0(1-2p_0)(1-2q_0)]$, and `pi2_0_0_0_0` represents $\mathbb{E}[p_0(1-p_0)q_0(1-q_0)]$. Here, there is only the one population (indexed by zero), but it should be clear how the indexing extends to additional populations.

One of the great strengths of `moments.LD` is that while it only computes low-order moments of the full two-locus haplotype distribution, it allows us to expand the basis of statistics to include many populations. For example, one of the example demographic models for two populations is `Demographics2D.split_mig`, in which a single population splits into two descendant populations, each with their own relative constant sizes and connected by symmetric migration.

```
y = moments.LD.Demographics2D.split_mig((0.5, 2.0, 0.2, 1.0), rho=1.0)
# here, the parameters of split_mig are (T, nu0, nu1, m_sym)
print(y.names())
y
```

```
(['DD_0_0', 'DD_0_1', 'DD_1_1', 'Dz_0_0_0', 'Dz_0_0_1', 'Dz_0_1_1', 'Dz_1_0_0', 'Dz_1_0_1
↪', 'Dz_1_1_1', 'pi2_0_0_0_0', 'pi2_0_0_0_1', 'pi2_0_0_1_1', 'pi2_0_1_0_1', 'pi2_0_1_1_1
↪', 'pi2_1_1_1_1'], ['H_0_0', 'H_0_1', 'H_1_1'])
```

```
LDstats([[8.56693276e-08 5.27620533e-08 8.04349065e-08 7.02703408e-08
  2.47530535e-08 5.40257063e-08 1.30712522e-07 4.83611466e-08
  5.90156000e-08 2.29151341e-07 2.86561358e-07 2.68396037e-07
  3.74454189e-07 3.53031720e-07 3.37988994e-07]], [0.00088909 0.00116239 0.00110805],␣
↪num_pops=2, pop_ids=None)
```

Notice that already with just two populations we pick up many additional statistics: not just $\mathbb{E}[D_0^2]$ and $\mathbb{E}[D_1^2]$, but also the cross population covariance of D: $\mathbb{E}[D_0 D_1]$, as well as all possible combinations of D, p, and q for the `Dz` and `pi2` moments. This is what makes such LD computation an efficient and powerful approach for inference: it is very fast to compute, it can be extended to many populations, and it gives us a large set informative statistics to compare to data and *run inference*.

5.1.1 LD decay curves

We are most often interested in examining how LD depends on recombination distances separating pairs of loci, given some underlying demography. Allele frequency correlations due to linkage are expected to break down faster with larger recombination distances, so that statistics such as D^2 decrease toward zero with increasing distances between SNPs.

In the literature, we typically see the decay of $r^2 = \mathbb{E}\left[\frac{D^2}{\pi_2}\right]$ or $\sigma_d^2 = \frac{\mathbb{E}[D^2]}{\mathbb{E}[\pi_2]}$ reported. These are related quantities, but there is a difference between the ratio of averages and the average of ratio. While solving for r^2 is very difficult, our `moments` framework immediately provides the expectations for σ_d^2 and other statistics of the same form (such as what we could call $\sigma_{Dz} = \frac{\mathbb{E}[Dz]}{\mathbb{E}[\pi_2]}$).

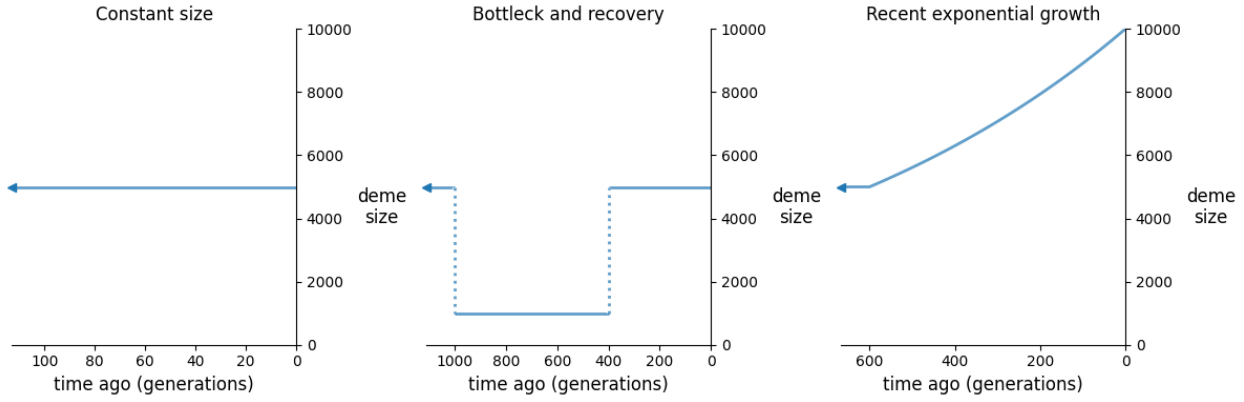
Here, we'll use `demes` to define a few simple models (which we'll illustrate with `demesdraw`), and explore how the decay of σ_d^2 and σ_{Dz} are affected by single-population demographic events. (Check out how to use Demes with moments.)

```
import demes, demesdraw
import matplotlib.pylab as plt

b1 = demes.Builder()
b1.add_deme(name="A", epochs=[dict(start_size=5000)])
```

(continues on next page)

(continued from previous page)

```
demog_constant = b1.resolve()

b2 = demes.Builder()
b2.add_deme(
    name="A",
    epochs=[
        dict(start_size=5000, end_time=1000),
        dict(start_size=1000, end_time=400),
        dict(start_size=5000, end_time=0)
    ]
)
demog_bottleneck = b2.resolve()

b3 = demes.Builder()
b3.add_deme(
    name="A",
    epochs=[dict(start_size=5000, end_time=600), dict(end_size=10000, end_time=0)]
)
demog_growth = b3.resolve()

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4))
demesdraw.size_history(demog_constant, ax=ax1, invert_x=True)
demesdraw.size_history(demog_bottleneck, ax=ax2, invert_x=True)
demesdraw.size_history(demog_growth, ax=ax3, invert_x=True)
ax1.set_ylim(top=10000)
ax2.set_ylim(top=10000)
ax3.set_ylim(top=10000)
ax1.set_title("Constant size")
ax2.set_title("Bottleck and recovery")
ax3.set_title("Recent exponential growth");
fig.tight_layout()
```



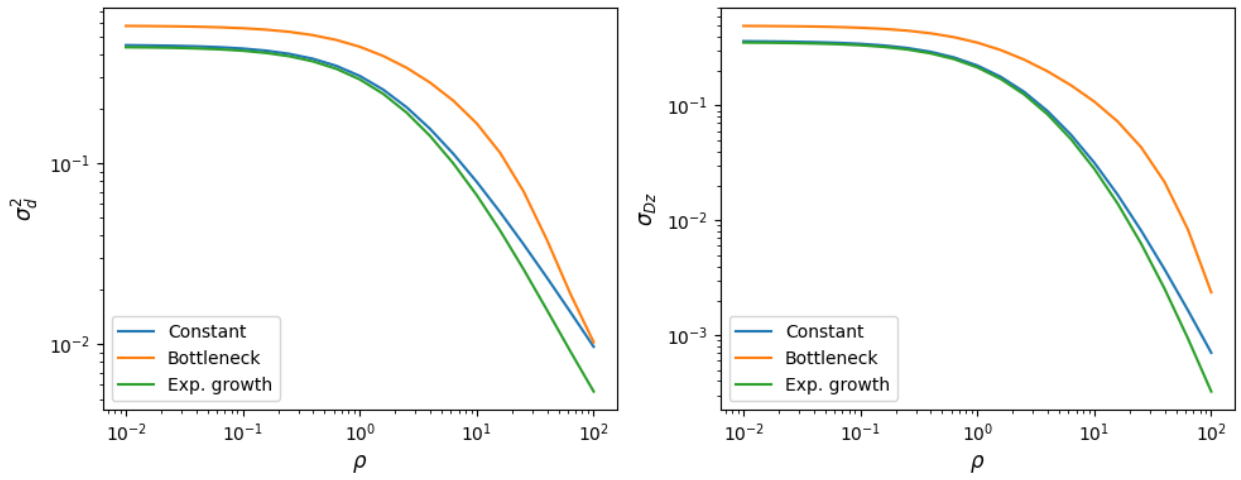
For each of these models, we'll compute LD statistics over a range of recombination rates, and then plot the decay curves.

```
import numpy as np

# set up recombination rates
rhos = np.logspace(-2, 2, 21)
```

(continues on next page)

(continued from previous page)

```
# compute statistics and normalize to get sigma-d^2 and sigma-Dz
y_constant = moments.Demes.LD(demog_constant, sampled_demes=["A"], rho=rhos)
sigma_constant = moments.LD.Inference.sigmaD2(y_constant)

y_bottleneck = moments.Demes.LD(demog_bottleneck, sampled_demes=["A"], rho=rhos)
sigma_bottleneck = moments.LD.Inference.sigmaD2(y_bottleneck)

y_growth = moments.Demes.LD(demog_growth, sampled_demes=["A"], rho=rhos)
sigma_growth = moments.LD.Inference.sigmaD2(y_growth)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

ax1.plot(rhos, sigma_constant.LD()[:, 0], label="Constant")
ax1.plot(rhos, sigma_bottleneck.LD()[:, 0], label="Bottleneck")
ax1.plot(rhos, sigma_growth.LD()[:, 0], label="Exp. growth")

ax2.plot(rhos, sigma_constant.LD()[:, 1], label="Constant")
ax2.plot(rhos, sigma_bottleneck.LD()[:, 1], label="Bottleneck")
ax2.plot(rhos, sigma_growth.LD()[:, 1], label="Exp. growth")

ax1.set_yscale("log")
ax2.set_yscale("log")
ax1.set_xscale("log")
ax2.set_xscale("log")
ax1.set_xlabel(r"$\rho$")
ax2.set_xlabel(r"$\rho$")
ax1.set_ylabel(r"$\sigma_d^2$")
ax2.set_ylabel(r"$\sigma_{Dz}$")
ax1.legend()
ax2.legend()
fig.tight_layout()
```



5.1.2 Multiple populations

The statistic $\mathbb{E}[D_i D_j]$, where i and j index two populations, is the covariance of LD between those populations. If these two population split from a common ancestral population, just after their split the covariance is equal to $\mathbb{E}[D^2]$ in the ancestral population. It then decays over time, to zero if there is no migration between them and to some positive value when they are connected by ongoing migration.

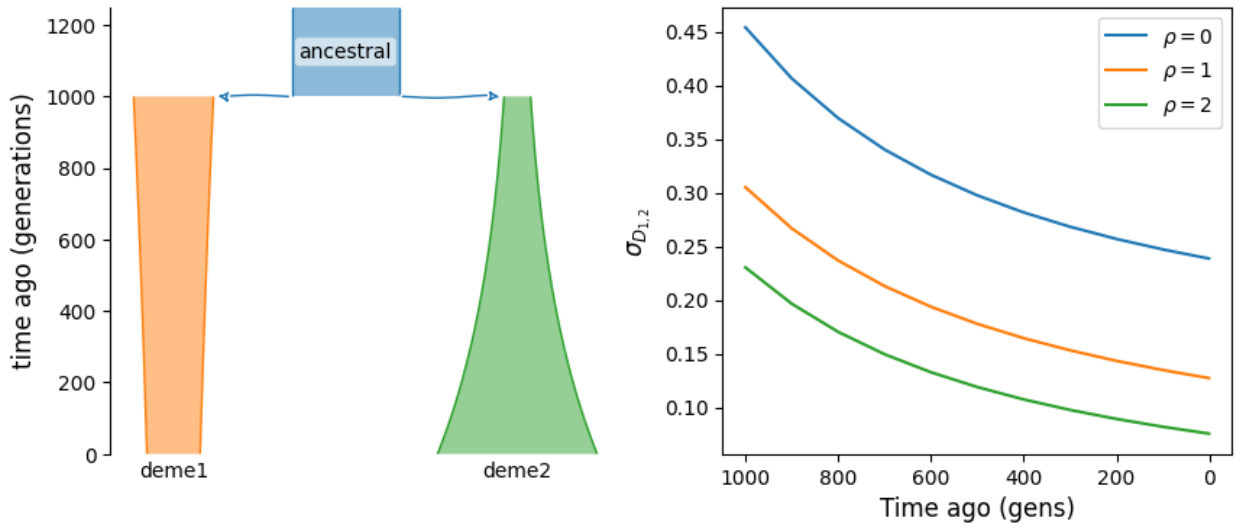
Here, we consider a simple split with isolation model and compute that covariance at different times in their history.

```
b = demes.Builder()
b.add_deme(name="ancestral", epochs=[dict(start_size=2000, end_time=1000)])
b.add_deme(
    name="deme1",
    ancestors=["ancestral"],
    epochs=[dict(start_size=1500, end_size=1000)]
)
b.add_deme(
    name="deme2",
    ancestors=["ancestral"],
    epochs=[dict(start_size=500, end_size=3000)]
)
g = b.resolve()

# get LD stats between deme1 and deme2 and times in the past, using ancient samples
ts = np.linspace(999, 1, 11, dtype="int")
rhos = [0, 1, 2]
def get_covD(g, ts, rhos):
    covD = {rho: [] for rho in rhos}
    for t in ts:
        y = moments.Demes.LD(
            g,
            sampled_demes=["deme1", "deme2"],
            sample_times=[t, t],
            rho=rhos
        )
        for rho in rhos:
            covD[rho].append(
                moments.LD.Inference.sigmaD2(y)[rhos.index(rho)][y.names()[0].index("DD_
↪0_1")]
            )
    return covD

covD = get_covD(g, ts, rhos)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

demesdraw.tubes(g, ax=ax1)

for rho in rhos:
    ax2.plot(ts, covD[rho], label=rf"$\rho={rho}$")

ax2.invert_xaxis()
ax2.set_xlabel("Time ago (gens)")
ax2.set_ylabel(r"$\sigma_{D_{1, 2}}$")
```

(continues on next page)

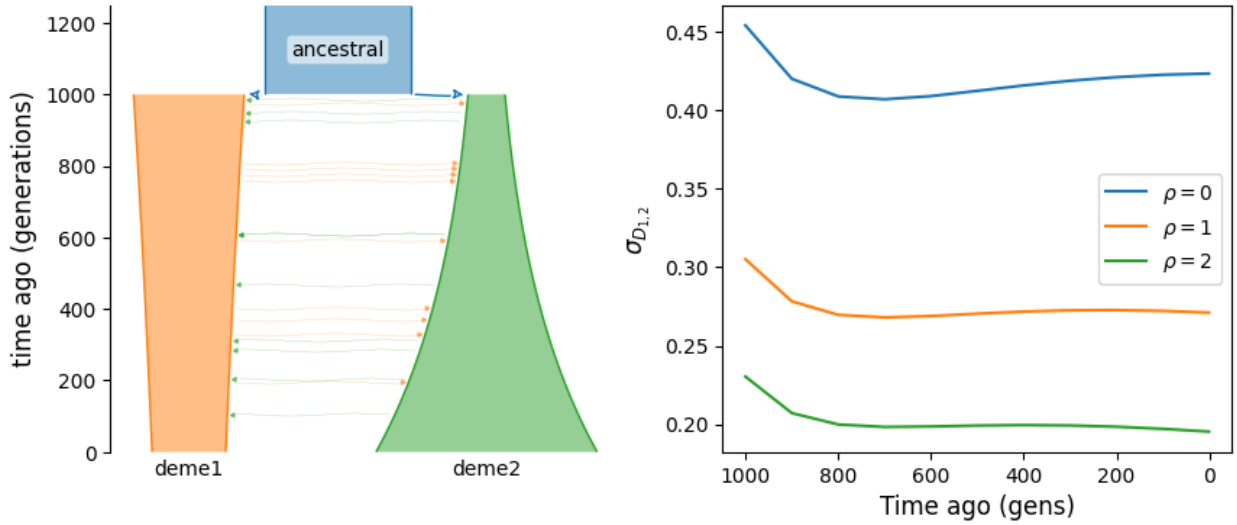
(continued from previous page)

```
ax2.legend();
```



We can see that without migration, covariance of LD across populations is expected to decay over time. If instead the two populations are connected by ongoing migration, LD will continue to have positive covariance, even long after their split from the ancestral population.

```
b.add_migration(demes=["deme1", "deme2"], rate=2e-3)
g = b.resolve()

covD = get_covD(g, ts, rhos)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

demesdraw.tubes(g, ax=ax1)

for rho in rhos:
    ax2.plot(ts, covD[rho], label=rf"$\rho={rho}$")

ax2.invert_xaxis()
ax2.set_xlabel("Time ago (gens)")
ax2.set_ylabel(r"$\sigma_{D_{1, 2}}$")
ax2.legend();
```



5.1.3 Archaic admixture

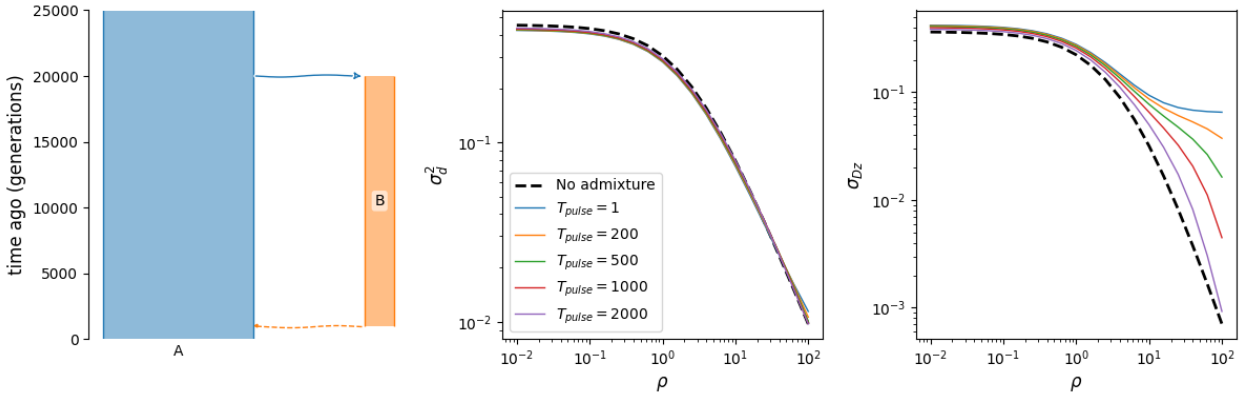
Finally, as shown in [Ragsdale2019], the σ_{Dz} statistic is particularly sensitive to archaic admixture. Unlike $\mathbb{E}[D^2]$, it is strongly elevated above single-ancestry expectations even with relatively small proportions of admixture from a deeply diverged source. Here, we have a very simple model of population that branches off from the focal population in the deep past and then provides 2% ancestry through admixture much more recently.

```
def admixture_model(t_pulse, prop=0.02):
    b = demes.Builder()
    b.add_deme(name="A", epochs=[dict(start_size=10000)])
    b.add_deme(
        name="B",
        ancestors=["A"],
        start_time=20000,
        epochs=[dict(start_size=2000, end_time=t_pulse)]
    )
    b.add_pulse(sources=["B"], dest="A", proportions=[prop], time=t_pulse)
    return b.resolve()

rhos = np.logspace(-2, 2, 21)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4))

demesdraw.tubes(admixture_model(1000), ax=ax1)

# without admixture
g = admixture_model(100, prop=0)
y = moments.Demes.LD(g, sampled_demes=["A"], rho=rhos)
sigma_d2 = moments.LD.Inference.sigmaD2(y)
ax2.plot(rhos, sigma_d2.LD()[:, 0], "k--", lw=2, label="No admixture")
ax3.plot(rhos, sigma_d2.LD()[:, 1], "k--", lw=2)

# varying admixture time
for t in [1, 200, 500, 1000, 2000]:
```

(continues on next page)

(continued from previous page)

```
    g = admixture_model(t)
    y = moments.Demes.LD(g, sampled_demes=["A"], rho=rhos)
    sigma_d2 = moments.LD.Inference.sigmaD2(y)
    ax2.plot(rhos, sigma_d2.LD()[:, 0], lw=1, label="$T_{pulse}=$"+f"${t}$")
    ax3.plot(rhos, sigma_d2.LD()[:, 1], lw=1)

ax2.legend()
ax2.set_xscale("log")
ax2.set_yscale("log")
ax3.set_xscale("log")
ax3.set_yscale("log")
ax2.set_xlabel(r"$\rho$")
ax3.set_xlabel(r"$\rho$")
ax2.set_ylabel(r"$\sigma_d^2$")
ax3.set_ylabel(r"$\sigma_{Dz}$")
fig.tight_layout();
```



5.2 Demographic events

As seen above, we can use either `demes` or the API to compute LD statistics under some demography. While `demes` is a very useful tool for building and visualizing demographic models, we sometimes want to use the built in functions to apply demographic events and integrate the LD stats object directly. Mirroring the `moments` API for manipulating SFS, we apply demographic events to LD objects using demographic functions that return a *new* LDstats object:

5.2.1 Extinction/marginalization

If a population goes extinct, or if we just want to stop tracking statistics involving that population, we can use `y.marginalize(idx)` to remove a given population or set of populations from the LD stats. Here, `idx` can be either an integer index or a list of integer indexes. `y.marginalize()` returns a new LD stats object with the specified populations removed and the population IDs preserved for the remaining populations (if given in the input LD stats).

5.2.2 Population splits

To split one population, we use `y.split(i, new_ids=["child1", "child2"])`, where `i` is the integer index of the population to split, and the optional argument `new_ids` lets us set the split population IDs. Note that if the input LD stats do not have population IDs defined (i.e `y.pop_ids == None`), we cannot specify new IDs.

5.2.3 Admixture and mergers

Admixture and merge events take two populations and combine them with given fractions of ancestry from each. The new admixed/merged population is placed at the end of the array of population indexes, and the only difference been `y.admix()` and `y.merge()` is that the `merge` function then removes the parental populations (i.e. the parents are marginalized after admixture).

For both functions, usage is `y.admix(idx0, idx1, f, new_id="xxx")`. We specify the indexes of the two parental populations (`idx0` and `idx1`) and the proportion `f` contributed by the first specified population `idx0` (population `idx1` contributes 1-`f`). We can also provide the ID of the admixed population using ``new_id`:

```
y = moments.LD.Demographics2D.snm(pop_ids=["A", "B"])
print(y.pop_ids)
y = y.admix(0, 1, 0.2, new_id="C")
print(y.pop_ids)
y = y.merge(1, 2, 0.75, new_id="D")
print(y.pop_ids)
```

```
['A', 'B']
['A', 'B', 'C']
['A', 'D']
```

5.2.4 Pulse migration

Finally, we can apply discrete (or pulse) mass migration events with a given proportion from one population to another. Here, we again specify 1) the index of the source population, 2) the index of the target/destination population, and 3) the proportion of ancestry contributed:

```
y = y.pulse_migrate(1, 0, 0.1)
print(y.pop_ids) # population IDs are unchanged.
```

```
['A', 'D']
```

5.3 Integration

Integrating the LD stats also mirrors the SFS integration function, with some changes to keyword arguments. At a minimum, we need to specify the relative sizes or size function `nu` and the integration time `T`. When simulating LD stats for one or more recombination rates, we also pass `rho` as a single rate or a list of rates, as needed:

```
y.integrate(nu, T, rho=rho, theta=theta)
```

For multiple populations, we can also specify a migration matrix of size $n \times n$, where n is the number of populations that the LD stats represents. Like the SFS integration, we can also specify any populations that are frozen by passing a list of length n with `True` for frozen populations and `False` for populations to integrate.

Unlike SFS integration, LD integration also lets us specify selfing rates within each population, where `selfing` is a list of length n that specifies the selfing rate within each deme, which must be between 0 and 1.

5.4 References

CHAPTER

SIX

PARSING LD STATISTICS

As described in the multi-population LD section, we are interested in σ_d^2-type statistics, which is the ratio of expectations of D^2 and $\pi_2 = p(1-p)q(1-q)$. Again, p and q are the allele frequencies at the left and right loci.

To estimate these statistics from data, we take the average of each LD statistic over all pairs of observed (biallelic) SNPs at a given recombination distance, and then divide by the observed π_2 in one of the populations (the "normalizing" population). As described below, we also use a block bootstrapping approach to estimate variances and covariances of observed statistics at each recombination distance, which is used in inference and computing confidence intervals.

6.1 Binned LD decay

To estimate LD decay curves from the data, we bin all pairs of observed SNPs by recombination distance. While we can bin by physical distance (bps) separating SNPs, genetic maps are non-uniform and physical distance does not perfectly correlate with genetic distance at small scales. If we have a recombination map available, it is preferable to compute and compare statistics using that map.

Recombination rate bins are defined by bin edges, which is a list or array with length equal to the number of desired bins plus one. Bin edges should be monotonically increasing, and are thus adjacent without gaps between bins. Thus, bins are defined as semi-open intervals:

```
import moments.LD
import numpy as np

bin_edges = np.array([0, 1e-6, 1e-5, 1e-4])
print("Bins:")
for b_l, b_r in zip(bin_edges[:-1], bin_edges[1:]):
    print(f"[{b_l}, {b_r})")
```

```
Bins:
[0.0, 1e-06)
[1e-06, 1e-05)
[1e-05, 0.0001)
```

There are a few considerations to keep in mind. In practice, very short distances can be problematic, because "non-standard" evolutionary processes can distort allele frequency correlations for tightly linked loci. For example, our evolutionary model does not include multi-nucleotide mutations [Harris2014] or gene conversion [Ardlie2001], both of which operate at short distances.

Thus, when working with real data we recommend omitting bins of very short recombination distances. In practice, we typically drop bins with length less than $r = 5 \times 10^{-6}$, which corresponds to roughly a few hundred bp on average in humans.

6.2 Parsing from a VCF

The primary function of the `Parsing` module is computing LD statistics from an input VCF. There are a number of options available, but the primary inputs are the path to the VCF file and the bins of distances separating loci. Typically, we work in recombination distance, in which case a recombination map is also required. If we do not have a recombination map available, we can bin by base pair distances instead.

The function `moments.LD.Parsing.compute_ld_statistics` returns a dictionary with the bins, returned statistics, populations, and *sums* of each statistic over the provided bins. For example:

```
r_bins = np.logspace(-6, -3, 7)
ld_stats = moments.LD.Parsing.compute_ld_statistics(
    vcf_path, r_bins=r_bins, rec_map_file=map_path)
```

6.2.1 Using a recombination map

The input recombination map is specified as a text file, with the first column giving positions along the chromosome and additional column(s) defining the cumulative map(s), typically in units of cM. The header line is "Pos Map1 Map2 (etc)", and we can use any map in the file by specifying the `map_name`. If no map name is given, or the specified map name does not match a genetic map in the header, we use the map in the first column.

Typically, maps are given in units of centi-Morgans, and the default behavior is to assume cM units. If the map is given in units of Morgans, we need to set `cM=False`.

6.2.2 Populations and pop-file

We often have data from more than one population, so we need to be able to specify which samples in the VCF correspond to which populations. This is handled by passing a file that assigns each sample to a population. For example, the population file is written as

```
sample  pop
sid_0   pop_A
sid_1   pop_B
sid_2   pop_A
sid_3   pop_A
sid_4   pop_B
...
```

Then to include the population information in the function, we also pass a list of the populations to compute statistics for. Samples from omitted populations are dropped from the data.

```
pops = ["pop_A", "pop_B"]
ld_stats = ld_stats = moments.LD.Parsing.compute_ld_statistics(
    vcf_path,
    r_bins=r_bins,
    rec_map_file=map_path,
    pop_file=pop_file_path,
    pops=pops
)
```

6.2.3 Masking and using bed files

If there are multiple chromosomes or contigs included in the VCF, we specify which chromosome to compute statistics for by setting the `chromosome` flag. We can also subset a chromosome by including a bed file, which will filter out all SNPs that fall outside the region intervals given in the bed file. Bed files have the format `{chrom}\t{left_pos}\t{right_pos}`, which defines a semi-open interval. The path to the bed file is provided with the `bed_file` argument.

6.2.4 Computing a subset of statistics

Sometimes we may wish to only compute a subset of possible LD statistics. By default, the parsing function computes all statistics possible for the number of populations provided. Instead, we can specify the `stats_to_compute`, which is a list (of length 2) of lists. The first list are the LD statistics to return, and the second list has the heterozygosity statistics to return. Statistic names follow the convention in `moments.LD.Util.moment_names(num_pops)`, and should be formatted accordingly.

6.2.5 Phased vs unphased data

We can compute LD statistics from either phased or unphased data. The default behavior is to assume that phasing is unknown, and `use_genotypes` is `True` by default. If we want to compute LD using phased data, we set `use_genotypes=False`, and parsing uses phased haplotypes instead. In general, phasing errors can bias LD statistics, sometimes significantly, and using genotypes instead of haplotypes only slightly increases uncertainty in most cases. Therefore, we usually recommend leaving `use_genotypes=True`.

6.3 Computing averages and covariances over regions

From `moments.LD.Parsing.compute_ld_statistics()`, we get LD statistic sums from the regions in a VCF, perhaps constrained by a bed file. Our strategy is to divide our data into some large number of roughly equally sized chunks, for example 500 regions across all 22 autosomes in human data. We then compute LD statistics independently for each region (it helps to parallelize that step, using a compute cluster). From those outputs, we can then compute average statistics genome-wide, as well as covariances of statistics within each bin. Those covariances are needed to be able to compute likelihoods and run optimization.

The outputs of `compute_ld_statistics` are compiled in a dictionary, where the keys are unique region identifiers, and items the outputs of that function. For example:

```
region_stats = {
    0: moments.LD.Parsing.compute_ld_statistics(VCF, bed_file="region_0.bed", ...),
    1: moments.LD.Parsing.compute_ld_statistics(VCF, bed_file="region_1.bed", ...),
    2: moments.LD.Parsing.compute_ld_statistics(VCF, bed_file="region_2.bed", ...),
    ...
}
```

Mean and variance-covariance matrices are computed by calling `bootstrap_data`, passing the region statistics dictionary, and optionally the index of the population to normalize σ_d^2 statistics by. By default, the normalizing population is the first (index 0).

```
mv = moments.LD.Parsing.bootstrap_data(region_stats)
```

`mv` contains the bins, statistics, and populations, as well as lists of mean statistics and variance-covariance matrices. This data can then be directly compared to model expectations and used in inference.

6.4 Example

Using msprime [Kelleher2016], we'll simulate some data under an isolation-with-migration (IM) model and then compute LD and heterozygosity statistics using the `LD.Parsing` methods. First, the simulation will use the `demes-msprime` interface, which are then written as a VCF.

The YAML-file specifying the model is

```
description: A simple isolation-with-migration model
time_units: generations
demes:
- name: anc
  epochs: [{start_size: 10000, end_time: 1500}]
- name: deme0
  ancestors: [anc]
  epochs: [{start_size: 2000}]
- name: deme1
  ancestors: [anc]
  epochs: [{start_size: 20000}]
migrations:
- demes: [deme0, deme1]
  rate: 1e-4
```

And we use msprime to simulate 1Mb of data, using a constant recombination and mutation rate.

```
import msprime
import demes
import os

# set up simulation parameters
L = 1e6
u = r = 1.5e-8
n = 10

g = demes.load("data/im-parsing-example.yaml")
demog = msprime.Demography.from_demes(g)

trees = msprime.sim_ancestry(
    {"deme0": n, "deme1": n},
    demography=demog,
    sequence_length=L,
    recombination_rate=r,
    random_seed=321,
)

trees = msprime.sim_mutations(trees, rate=u, random_seed=123)

with open("data/im-parsing-example.vcf", "w+") as fout:
    trees.write_vcf(fout)
```

This simulation had 10 diploid individuals per population, and `msprime/tskit` writes their IDs as `tsk_0`, `tsk_1`, etc:

```
##fileformat=VCFv4.2
##source=tskit 0.3.5
```

(continues on next page)

(continued from previous page)

```
##FILTER=<ID=PASS,Description="All filters passed">
##contig=<ID=1,length=10000000>
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">

↪#CHROM      POS       ID        REF       ALT       QUAL      FILTER      INFO        FORMAT
↪0       tsk_1       tsk_2       tsk_3       tsk_4       tsk_5       tsk_
↪6       tsk_7       tsk_8       tsk_9       tsk_10       tsk_11       tsk_
↪12       tsk_13       tsk_14       tsk_15       tsk_16       tsk_17       tsk_
↪18       tsk_19
1       221       .       A       T       .       PASS       .
↪       GT       1|1       1|0       1|1       1|1       0|1       0|1       0|0       1|1
1       966       .       A       C       .       PASS       .
↪       GT       0|0       0|0       0|0       0|0       0|0       0|0       0|0       1|1
1       1082       .       G       A       .       PASS       .
↪       GT       0|0       0|1       0|0       0|0       1|0       1|0       1|1       0|0
1       1133       .       G       T       .       PASS       .
↪       GT       0|0       0|1       0|0       0|0       1|0       1|0       1|1       0|0
```

To parse this data, we need the file that maps samples to populations and the recombination map file (the total map length is found by 1×10^6 bp $\times$ 1.5×10^{-8} M/bp $\times$ 100 cM/M):

```
sample  pop
tsk_0   deme0
tsk_1   deme0
tsk_2   deme0
tsk_3   deme0
tsk_4   deme0
tsk_5   deme0
tsk_6   deme0
tsk_7   deme0
tsk_8   deme0
tsk_9   deme0
tsk_10  deme1
tsk_11  deme1
tsk_12  deme1
tsk_13  deme1
tsk_14  deme1
tsk_15  deme1
tsk_16  deme1
tsk_17  deme1
tsk_18  deme1
tsk_19  deme1
```

```
Pos Map(cM)
0   0
1000000 1.5
```

With all this, we can now compute LD based on recombination distance bins:

```
r_bins = np.array(
    [0, 1e-6, 2e-6, 5e-6, 1e-5, 2e-5, 5e-5, 1e-4, 2e-4, 5e-4, 1e-3]
)
```

(continues on next page)

(continued from previous page)

```
vcf_file = "data/im-parsing-example.vcf"
map_file = "data/im-parsing-example.map.txt"
pop_file = "data/im-parsing-example.samples.pops.txt"
pops = ["deme0", "deme1"]
ld_stats = moments.LD.Parsing.compute_ld_statistics(
    vcf_file,
    rec_map_file=map_file,
    pop_file=pop_file,
    pops=["deme0", "deme1"],
    r_bins=r_bins,
    report=False,
)
```

The output, `ld_stats`, is a dictionary with the keys `bins`, `stats`, `pops`, and `sums`. To get the average statistics over multiple regions (here, we only have a single region that we simulated), we use `means_from_region_data`:

```
means = moments.LD.Parsing.means_from_region_data(
    {0: ld_stats}, ld_stats["stats"], norm_idx=0
)
```

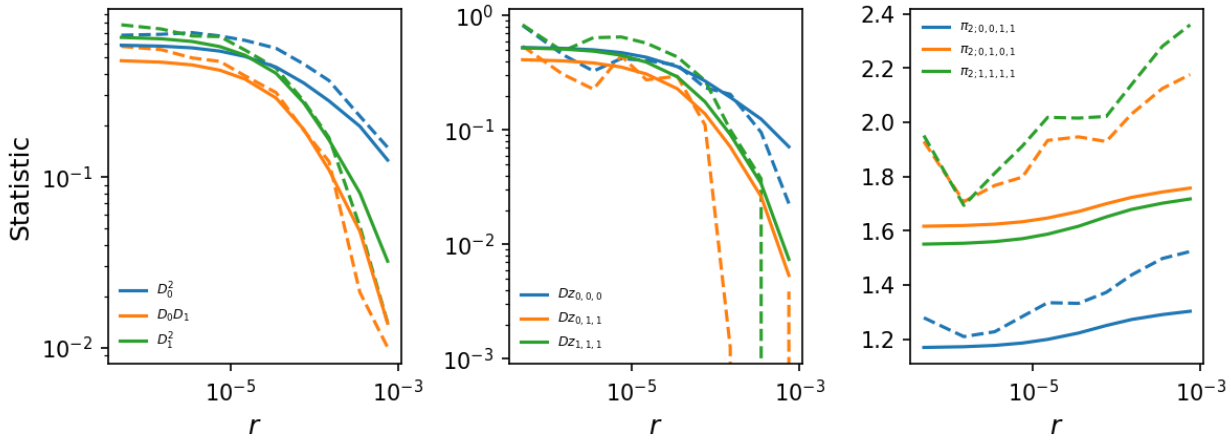
This provides σ_d^2-type statistics relative to *pi_2* in `deme0`, and relative heterozygosities (also relative to `deme0`). These statistics were computed from only a single relatively small region, so they will be quite noisy. But we can still compare to expectations under the input IM demographic model.

```
import demes
g = demes.load("data/im-parsing-example.yaml")

y = moments.Demes.LD(
    g,
    sampled_demes=["deme0", "deme1"],
    rho=4 * g["anc"].epochs[0].start_size * r_bins,
)

# stats are computed at the bin edges - average to get midpoint estimates
y = moments.LD.LDstats(
    [(y_l + y_r) / 2 for y_l, y_r in zip(y[:-2], y[1:-1])] + [y[-1]],
    num_pops=y.num_pops,
    pop_ids=y.pop_ids,
)
y = moments.LD.Inference.sigmaD2(y)

# plot LD decay curves for some statistics
moments.LD.Plotting.plot_ld_curves_comp(
    y,
    means[:-1],
    [],
    rs=r_bins,
    stats_to_plot=[
        ["DD_0_0", "DD_0_1", "DD_1_1"],
        ["Dz_0_0_0", "Dz_0_1_1", "Dz_1_1_1"],
        ["pi2_0_0_1_1", "pi2_0_1_0_1", "pi2_1_1_1_1"]
```

(continues on next page)

(continued from previous page)

```
    ],
    labels=[[r"$D_0^2$", r"$D_0 D_1$", r"$D_1^2$"],
        [r"$Dz_{0,0,0}$", r"$Dz_{0,1,1}$", r"$Dz_{1,1,1}$"],
        [r"$\pi_{2;0,0,1,1}$", r"$\pi_{2;0,1,0,1}$", r"$\pi_{2;1,1,1,1}$"]
    ],
    plot_vcs=False,
    fig_size=(8, 3),
    show=True,
)
```



6.4.1 Bootstrapping over multiple regions

Normally, we'll want more data than from a single 1Mb region to compute averages and variances of statistics. Using the same approach as the above example, `ld_stats` for 100 replicates we computed (see example in the `moments` repository here). From this, each replicate set of statistics were placed in a dictionary, as `rep_stats = {0: ld_stats_0, 1: ld_stats_1, ..., 99: ld_stats_99}`. This dictionary can then be used to compute means and covariances of statistics.

```
mv = moments.LD.Parsing.bootstrap_data(ld_stats)
```

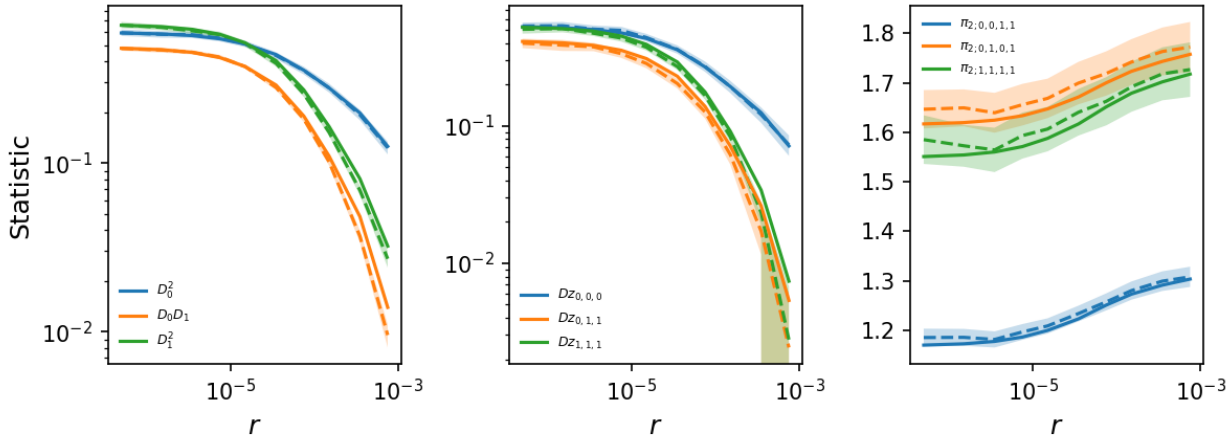
By simulating more data, the LD decay curves are much less noisy, and by simulating multiple replicates, we also compute the variance-covariance matrices for each bin and can include standard errors in the plots.

```
# plot LD decay curves for some statistics
moments.LD.Plotting.plot_ld_curves_comp(
    y,
    mv["means"][:-1],
    mv["varcovs"][:-1],
    rs=r_bins,
    stats_to_plot=[
        ["DD_0_0", "DD_0_1", "DD_1_1"],
        ["Dz_0_0_0", "Dz_0_1_1", "Dz_1_1_1"],
        ["pi2_0_0_1_1", "pi2_0_1_0_1", "pi2_1_1_1_1"]
    ],
    labels=[[r"$D_0^2$", r"$D_0 D_1$", r"$D_1^2$"],
        [r"$Dz_{0,0,0}$", r"$Dz_{0,1,1}$", r"$Dz_{1,1,1}$"],
```

(continues on next page)

(continued from previous page)

```
        [r"$\pi_{2;0,0,1,1}$", r"$\pi_{2;0,1,0,1}$", r"$\pi_{2;1,1,1,1}$"]
    ],
    plot_vcs=True,
    fig_size=(8, 3),
    show=True,
)
```



Note: The means-covariances data is required for inference using LD statistics. In *Inferring demography with LD*, we'll use the same `mv` data dictionary to refit the IM model as an example.

6.5 LD statistics in genotype blocks

`moments.LD.Parsing` also includes some functions for computing LD from genotype (or haplotype) blocks. Genotype blocks are arrays of shape $L \times n$, where L is the number of loci and n is the sample size. We assume a single population, and so we compute D^2, Dz, π_2, and D, either pairwise or averaged over all pairwise comparisons.

If we have a genotype matrix containing n diploid samples, genotypes are coded as 0, 1, and 2, and we set `genotypes=True`. If we have a haplotype matrix with data from n haploid copies, genotypes are coded as 0 and 1 only, and we set `genotypes=False`.

For example, given a single genotype matrix, we compute all pairwise statistics and average statistics as shown below:

```
L = 10
n = 5
G = np.random.randint(3, size=L * n).reshape(L, n)

# all pairwise comparisons:
D2_pw, Dz_pw, pi2_pw, D_pw = moments.LD.Parsing.compute_pairwise_stats(G)

# averages:
D2_ave, Dz_ave, pi2_ave, D_ave = moments.LD.Parsing.compute_average_stats(G)
```

Similarly, we can compute the pairwise or average statistics between two genotype matrices. The matrices can have differing number of loci, but they must have the same number of samples, as the genotype matrices are assumed to come from different regions within the same samples.

```
L2 = 12
n = 5

G2 = np.random.randint(3, size=L2 * n).reshape(L2, n)

# all pairwise comparisons:
D2_pw, Dz_pw, pi2_pw, D_pw = moments.LD.Parsing.compute_pairwise_stats_between(G, G2)

# averages:
D2_ave, Dz_ave, pi2_ave, D_ave = moments.LD.Parsing.compute_average_stats_between(G, G2)
```

Note: Computing LD in genotype blocks uses C-extensions that are not built by default, so are only available if these are built when compiling the C-extensions. In order to use these methods, we need to build these extensions using the `--ld_extensions` flag, as `python setup.py build_ext --ld_extensions -i`.

6.6 References

CHAPTER

SEVEN

INFERRING DEMOGRAPHY WITH LD

As described in the *linkage disequilibrium* and *LD Parsing* sections, we use a family of normalized LD and heterozygosity statistics to compare between model expectations and data. We optimize demographic model parameters to find the expected binned LD and heterozygosity statistics that maximize a composite likelihood over all pairs of SNPs and recombination bins.

In this section, we'll describe the likelihood framework, how to define demographic models that can be used in inference, how to run optimization using `moments`' built-in inference functions, and how to compute confidence intervals. We include a short example, following the *parsing* of data simulated under an isolation-with-migration model, to illustrate the main features and options.

7.1 Likelihood framework

For a given recombination distance bin indexed by i, we have a set of computed LD statistic means D_i from data along with the variance-covariance matrix Σ_i as returned by `moments.LD.Parsing.bootstrap_data`. We assume a multivariate Guassian likelihood function, so that a model parameterized by Θ that has expected statistics $\mu_i(\Theta)$ has likelihood

$$\mathcal{L}_i(\Theta|D_i) = P(D_i|\mu_i, \Sigma_i) = \frac{\exp\left(-\frac{1}{2}(D_i - \mu_i)^T\Sigma_i^{-1}(D_i - \mu_i)\right)}{(2\pi)^{k/2}|\Sigma_i|^{1/2}}.$$

The likelihood is computed similarly for heterozygosity statistics, given their variance-covariance matrix. Then the composite likelihood of two-locus data across recombination bins and single-locus heterozygosity (indexed by $i = n+1$ where n is the total number of recombination bins), is

$$\mathcal{L} = \prod_{i=1}^{n+1} \mathcal{L}_i.$$

In practice, we work with the log of the likelihood, so that products turn to sums and we can drop constant factors:

$$\log \mathcal{L} \propto -\frac{1}{2}\sum_{i=1}^{n+1}(D_i - \mu_i)^T\Sigma_i^{-1}(D_i - \mu_i).$$

As the data $\{D_i, \Sigma_i\}$ is fixed, we search for the model parameters Θ that provide $\{\mu_i\}$ that maximizes $\log \mathcal{L}$.

7.2 Defining demographic models

There are a handful of built-in demographic models for one-, two-, and three-population scenarios that can be used in inference (see *here*). However, these are far from comprehensive and it is likely that custom demographic models will need to be written for a given inference problem. For inspiration, `moments.LD.Demographics1D`, `Demographics2D`, and `Demographics3D` can be used as starting points and as illustrations of how to structure model functions.

Demographic models all require a `params` positional argument and `rho` and (optionally) `theta` keyword arguments. `theta`, the population-size scaled mutation rate, does not play a role in inference using relative statistics, as the mutation rate cancels in σ_d^2-type statistics.

For example, the IM model we simulated data under in the LD Parsing section could be parameterized as

```
def model_func(params, rho=None, theta=0.001):
    nu0, nu1, T, M = params
    y = moments.LD.Numerics.steady_state(rho=rho, theta=theta)
    y = moments.LD.LDstats(y, num_pops=1)
    y = y.split(0)
    y.integrate([nu0, nu1], T, m=[[0, M], [M, 0]], rho=rho, theta=theta)
    return y
```

In the input demographic model to the simulations, we had the ancestral effective population size as 10,000, the size of deme0 was 2,000, and the size of deme1 was 20,000. The populations split 1,500 generations ago, and exchanged migrants symmetrically at a rate of 0.0001 per-generation. Converted into genetic units, `nu0 = 0.2`, `nu1 = 2`, `T=1500 / 2 / 10000 = 0.075`, and `M = 2 * 10000 * 0.0001 = 2.0`.

7.3 Running optimization

Optimization with `moments.LD`, much like `moments` optimization with the SFS, includes a handful functions that serve as wrappers for `scipy` optimizers with options specific to working with LD statistics. The two primary functions in `moments.LD.Inference` are

- `optimize_log_fmin`: Uses the downhill simplex algorithm on the log of the parameters.
- `optimize_log_powell`: Uses the modified Powell's method, which optimizes slices of parameter space sequentially.

Each optimization method accepts the same arguments. Required positional arguments are

- `p0`: The initial guess for the parameters in `model_func`.
- `data`: Structured as a list of lists of data means and data var-cov matrices. I.e., `data = [[means[0], means[1], ...], [varcovs[0], varcovs[1], ...]]`, with the final entry of the lists the means and var-covs of the heterozygosity statistics.
- `model_func`: The demographic model to be fit (see above section). Importantly, this is a *list*, where the first entry is the LD model, which is always used, and the optional second entry is a demographic model for the SFS (which is a rarely used option and can be ignored). So usually, we would set `model_func` as `[model_func_ld]`.

Additionally, we will almost always pass the list of unscaled recombination bin edges as `rs = [r0, r1, ..., rn]`, which defines *n* recombination bins.

The effective population size plays a different role in LD inference than it does in SFS inference. For the site frequency spectrum, N_e merely acts as a linear scaling factor and is absorbed by the scaled mutation rate θ, which is treated as a free parameter. Here, N_e instead rescales recombination rates, and because we use a recombination map to determine the binning of data by recombination distances separating loci, N_e is a parameter that must be either passed as a fixed value or simultaneously fit in the optimization.

If `Ne` is a fixed value, we specify the population size using that keyword argument. Otherwise, if `Ne` is to be fit, our list of parameters to fit by convention includes `Ne` in the final position in the list. Typically, `Ne` is not a parameter of the demographic model, as we work in rescaled genetic units, so the parameters that get passed to `model_func` are `params[:-1]`. However, it is also possible to write a demographic model that also uses `Ne` as a parameter. In this case we set `pass_Ne` to `True`, so that `Ne` both rescales recombination rates and is a model parameter, and all `params` are passed to `model_func`.

- `Ne`: The effective population size, used to rescale `rs` to get `rhos = 4 * Ne * rs`.
- `pass_Ne`: Defaults to `False`. If `True`, the demographic model includes `Ne` as a parameter (in the final position of input parameters).

Other commonly used options include

- `fixed_params`: Defaults to `None`. To fix some parameters, this should be a list of equal length as `p0`, with `None` for parameters to be fit and fixed values at corresponding indexes.
- `lower_bound`: Defaults to `None`. Constraints on the lower bounds during optimization. These are given as lists of the same length of the parameters.
- `upper_bound`: Defaults to `None`. Constraints on the upper bounds during optimization. These are given as lists of the same length of the parameters.
- `statistics`: Defaults to `None`, which assumes that all statistics are present and in the conventional default order. If the data is missing some statistics, we must specify which statistics are present using the subset of statistic names given by `moments.LD.Util.moment_names(num_pops)`.
- `normalization`: Defaults to `0`. The index of the population to normalize by, which should match the population index that we normalized by when parsing the data.
- `verbose`: If an integer greater than 0, prints updates of the optimization procedure at intervals given by that spacing.

7.3.1 Example

Using the data simulated in the *Parsing* section, we can refit the demographic model under a parameterized IM model. For this, we could use the `moments.LD.Demographics2D.split_mig` model as our `model_func`, which is equivalent to the function we defined above (which we use in this example). After loading the data and setting up the inference options, we'll use `optimize_log_fmin` to fit the model.

```
import moments.LD
import pickle

data = pickle.load(open("data/means.varcovs.split_mig.100_reps.bp", "rb"))

rs = [0, 1e-6, 2e-6, 5e-6, 1e-5, 2e-5, 5e-5, 1e-4, 2e-4, 5e-4, 1e-3]

p_guess = [0.1, 2.0, 0.075, 2.0, 10000]
p0 = moments.LD.Util.perturb_params(p_guess, fold=0.2)

# run optimization
opt_params, LL = moments.LD.Inference.optimize_log_fmin(
    p_guess,
    [data["means"], data["varcovs"]],
    [model_func],
    rs=rs,
    verbose=40,
```

(continues on next page)

(continued from previous page)

```
)

# get physical units, rescaling by Ne
physical_units = moments.LD.Util.rescale_params(
    opt_params, ["nu", "nu", "T", "m", "Ne"]
)

print("best fit parameters:")
print(f"  N(deme0)          :  {physical_units[0]:.1f}")
print(f"  N(deme1)          :  {physical_units[1]:.1f}")
print(f"  Div. time (gen)   :  {physical_units[2]:.1f}")
print(f"  Migration rate    :  {physical_units[3]:.6f}")
print(f"  N(ancestral)      :  {physical_units[4]:.1f}")
```

```
40      , -214.296    , array([ 0.140269   ,  1.95203    ,  0.0514585  ,  2.07543    , ␣
↪12430      ])
```

```
80      , -130.758    , array([ 0.153478   ,  2.00214    ,  0.0541354  ,  2.11845    , ␣
↪11248.2    ])
```

```
120     , -77.1877    , array([ 0.184635   ,  1.80666    ,  0.0673507  ,  2.0472     , ␣
↪11068.5    ])
```

```
160     , -76.8557    , array([ 0.186388   ,  1.80029    ,  0.0685854  ,  2.05002    , ␣
↪11008.4    ])
```

```
200     , -76.823     , array([ 0.185232   ,  1.80143    ,  0.0679905  ,  2.04797    , ␣
↪11010.4    ])
```

```
240     , -76.8106    , array([ 0.185808   ,  1.80374    ,  0.0683159  ,  2.05049    , ␣
↪11000.2    ])
```

```
280     , -76.4522    , array([ 0.188075   ,  1.81258    ,  0.0692172  ,  2.02497    , ␣
↪10970.3    ])
```

```
320     , -74.2099    , array([ 0.183888   ,  1.92246    ,  0.0656908  ,  1.82861    , ␣
↪10918.1    ])
```

```
360     , -72.9169    , array([ 0.191533   ,  2.01901    ,  0.0680872  ,  1.70206    , ␣
↪10638.2    ])
```

```
400     , -72.7291    , array([ 0.192878   ,  2.06349    ,  0.0681534  ,  1.66157    , ␣
↪10576      ])
```

```
440     , -72.7222    , array([ 0.193385   ,  2.06172    ,  0.0684375  ,  1.67061    , ␣
↪10567.8    ])
```

```
best fit parameters:
  N(deme0)          :  2042.8
```

(continues on next page)

(continued from previous page)

```
N(deme1)          :  21785.4
Div. time (gen)   :  1445.7
Migration rate    :  0.000079
N(ancestral)      :  10569.8
```

These should be pretty close to the input demographic parameters from the simulations! They won't be spot on, as this was only using 100Mb of simulated data, but we should be in the ballpark.

7.4 Computing confidence intervals

When running demographic inference, we get a point estimate for the *best fit* demographic parameters. However, for an unknown underlying true value, it's important to also estimate what's called a confidence interval. The CI tells us the probability that the true value lies within some range, and provides some information about which parameters in our demographic model are tightly constrained and which parameters we have little power to pin down.

`moments.LD` can estimate confidence intervals using either the Fisher Information Matrix (FIM) or the Godambe Information Matrix (GIM). In almost all cases when using real data (or even most simulated data), the FIM will estimate a much smaller CI than the GIM. This occurs because the FIM assumes all data points that we've used are independent, when in reality there is linkage that causes data points to be sometimes highly correlated between pairs of loci and between recombination bins. The Godambe method uses bootstrap-resampled replicates of the data to account for this correlation and does a much better job at estimating the true underlying CIs [Coffman2016].

Note: If you use the Godambe approach to estimate confidence intervals, please cite [Coffman2016]. Alec originally implemented this approach in `dadi`, and `moments` has more-or-less used this same implementation here.

To create bootstrap replicates from the dictionary of data sums computed over regions, where `rep_data = {0: ld_stats_0, 1: ld_stats_1, ...}`, e.g., we use

```
num_boots = 100
norm_idx = 0
bootstrap_sets = moments.LD.Parsing.get_bootstrap_sets(
    rep_data, num_bootstraps=num_boots, normalization=norm_idx)
```

These bootstrap sets can then be used as the inputs to the `moments.LD.Godambe` methods. The two CI estimation methods are

- `FIM_uncert`: Uses the Fisher Information Matrix. Usage is `FIM_uncert(model_func, opt_params, means, varcovs, r_edges=rs)`.
- `GIM_uncert`: Uses the Godambe Information Matrix. Usage is `GIM_uncert(model_func, bootstrap_sets, opt_params, means, varcovs, r_edges=rs)`.

In each case, the model function is the same as used in inference (some manipulation may be needed if we had any fixed parameters), means and varcovs are the same data as input to the inference function, and `r_edges` are the bin edges used in the inference. Additional options for some corner cases are described in the *API reference for LD methods*.

7.4.1 Example

We'll use both the FIM and GIM to compute uncertainties from the above example inference.

Using the FIM approach:

```
# using FIM
uncerts_FIM = moments.LD.Godambe.FIM_uncert(
    model_func,
    opt_params,
    data["means"],
    data["varcovs"],
    r_edges=rs,
)

# lower and upper CIs, in genetic units
lower = opt_params - 1.96 * uncerts_FIM
upper = opt_params + 1.96 * uncerts_FIM

# convert to physical units
lower_pu = moments.LD.Util.rescale_params(lower, ["nu", "nu", "T", "m", "Ne"])
upper_pu = moments.LD.Util.rescale_params(upper, ["nu", "nu", "T", "m", "Ne"])

print("95% CIs:")
print(f"  N(deme0)          :  {lower_pu[0]:.1f} - {upper_pu[0]:.1f}")
print(f"  N(deme1)          :  {lower_pu[1]:.1f} - {upper_pu[1]:.1f}")
print(f"  Div. time (gen)   :  {lower_pu[2]:.1f} - {upper_pu[2]:.1f}")
print(f"  Migration rate    :  {lower_pu[3]:.6f} - {upper_pu[3]:.6f}")
print(f"  N(ancestral)      :  {lower_pu[4]:.1f} - {upper_pu[4]:.1f}")
```

```
95% CIs:
  N(deme0)          :  1828.5 - 2267.9
  N(deme1)          :  18863.7 - 24873.3
  Div. time (gen)   :  1269.6 - 1631.3
  Migration rate    :  0.000069 - 0.000088
  N(ancestral)      :  10165.2 - 10974.3
```

And using the GIM approach:

```
bootstrap_sets = pickle.load(open("data/bootstrap_sets.split_mig.100_reps.bp", "rb"))

# using GIM
uncerts_GIM = moments.LD.Godambe.GIM_uncert(
    model_func,
    bootstrap_sets,
    opt_params,
    data["means"],
    data["varcovs"],
    r_edges=rs,
)

# lower and upper CIs, in genetic units
lower = opt_params - 1.96 * uncerts_GIM
upper = opt_params + 1.96 * uncerts_GIM
```

(continues on next page)

(continued from previous page)

```
# convert to physical units
lower_pu = moments.LD.Util.rescale_params(lower, ["nu", "nu", "T", "m", "Ne"])
upper_pu = moments.LD.Util.rescale_params(upper, ["nu", "nu", "T", "m", "Ne"])

print("95% CIs:")
print(f"  N(deme0)         :  {lower_pu[0]:.1f} - {upper_pu[0]:.1f}")
print(f"  N(deme1)         :  {lower_pu[1]:.1f} - {upper_pu[1]:.1f}")
print(f"  Div. time (gen)  :  {lower_pu[2]:.1f} - {upper_pu[2]:.1f}")
print(f"  Migration rate   :  {lower_pu[3]:.6f} - {upper_pu[3]:.6f}")
print(f"  N(ancestral)     :  {lower_pu[4]:.1f} - {upper_pu[4]:.1f}")
```

```
95% CIs:
  N(deme0)         :  1581.2 - 2551.3
  N(deme1)         :  17105.0 - 26931.2
  Div. time (gen)  :  1030.5 - 1906.9
  Migration rate   :  0.000059 - 0.000096
  N(ancestral)     :  9854.5 - 11285.0
```

We can see above that the FIM uncertainties are considerably smaller (i.e. more constrained) than the GIM uncertainties. However, the GIM uncertainties are to be preferred here, as they more accurately estimate the underlying true uncertainty in the demographic inference.

7.5 References

CHAPTER

EIGHT

SPECIFYING MODELS WITH `demes`

New in version 1.1, `moments` can compute the SFS and LD statistics directly from a `demes`-formatted demographic model. To learn about how to describe a demographic model using `demes`, head to the demes repository or documentation to learn about specifying multi-population demographic models using `demes`.

8.1 What is `demes`?

Demographic models specify the historical size changes, migrations, splits and mergers of related populations. Specifying demographic models using `moments` or practically any other simulation engine can become very complicated and error prone, especially when we want to model more than one population (e.g. [Ragsdale2020]). Even worse, every individual software has its own language and methods for specifying a demographic model, so a user has to reimplement the same model across multiple software, which nobody enjoys. To resolve these issues of reproducibility, replication, and susceptibility to errors, `demes` provides a human-readable specification of complex demography that is designed to make it easier to implement and share models and to be able to use that demography with multiple simulation engines.

`Demes` models are written in YAML, and they are then automatically parsed to create an internal representation of the demography that is readable by `moments`. `moments` can then iterate through the epochs and demographic events in that model and compute the SFS or LD.

8.2 Simulating the SFS and LD using a `demes` model

Computing expectations for the SFS or LD using a `demes` model is designed to be as simple as possible. In fact, there is no need for the user to specify any demographic events or integrate the SFS or LD objects. `moments` does all of that for you.

It's easiest to see the functionality through example. In the tests directory, there is a YAML description of the [Gutenkunst2009] Out-of-African model:

```
description: The Gutenkunst et al. (2009) three-population model of human history.
doi:
  - https://doi.org/10.1371/journal.pgen.1000695
time_units: years
generation_time: 25
demes:
  - name: ancestral
    description: Equilibrium/root population
    epochs:
    - end_time: 220e3
      start_size: 7300
```

(continues on next page)

(continued from previous page)

```
  - name: AMH
    description: Anatomically modern humans
    ancestors: [ancestral]
    epochs:
    - end_time: 140e3
      start_size: 12300
  - name: OOA
    description: Bottleneck out-of-Africa population
    ancestors: [AMH]
    epochs:
    - end_time: 21.2e3
      start_size: 2100
  - name: YRI
    description: Yoruba in Ibadan, Nigeria
    ancestors: [AMH]
    epochs:
    - start_size: 12300
      end_time: 0
  - name: CEU
    description: Utah Residents (CEPH) with Northern and Western European Ancestry
    ancestors: [OOA]
    epochs:
    - start_size: 1000
      end_size: 29725
      end_time: 0
  - name: CHB
    description: Han Chinese in Beijing, China
    ancestors: [OOA]
    epochs:
    - start_size: 510
      end_size: 54090
      end_time: 0
migrations:
  - demes: [YRI, OOA]
    rate: 25e-5
  - demes: [YRI, CEU]
    rate: 3e-5
  - demes: [YRI, CHB]
    rate: 1.9e-5
  - demes: [CEU, CHB]
    rate: 9.6e-5
```

This model describes all the populations (demes), their sizes and times of existence, their relationships to other demes (ancestors and descendents), and migration between them. To simulate using this model, we just need to specify the populations that we want to sample lineages from, the sample size in each population, and (optionally) the time of sampling. If sampling times are not given we assume we sample at present time. Ancient samples can be specified by setting sampling times greater than 0.

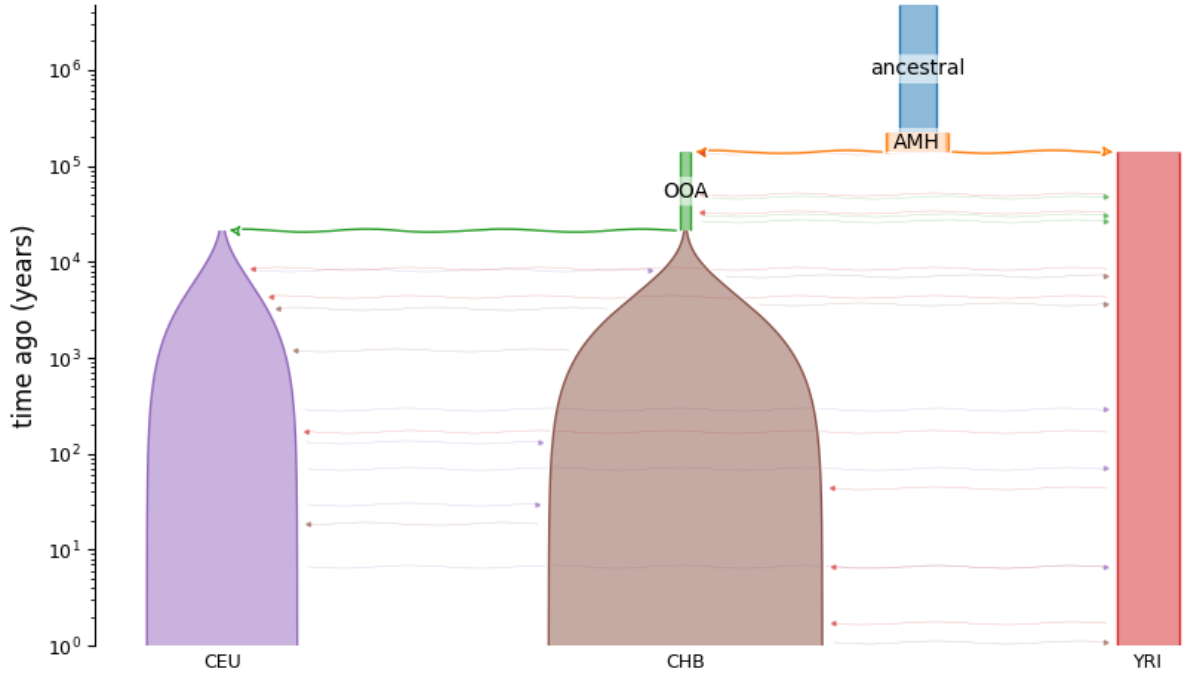
Let's simulate 10 samples from each YRI, CEU, and CHB:

```
import moments
import numpy as np
ooa_model = "../tests/test_files/gutenkunst_ooa.yaml"
```

(continues on next page)

(continued from previous page)

```
# we can visualize the model using demesdraw
import demes, demesdraw, matplotlib.pylab as plt
graph = demes.load(ooa_model)
demesdraw.tubes(graph, log_time=True, num_lines_per_migration=3);
```



Let's simulate 10 samples from each YRI, CEU, and CHB:

```
sampled_demes = ["YRI", "CEU", "CHB"]
sample_sizes = [10, 10, 10]

fs = moments.Spectrum.from_demes(
    ooa_model, sampled_demes=sampled_demes, sample_sizes=sample_sizes
)

print("populations:", fs.pop_ids)
print("sample sizes:", fs.sample_sizes)
print("FST:")
for k, v in fs.Fst(pairwise=True).items():
    print(f"  {k[0]}, {k[1]}: {v:.3f}")
```

```
populations: ['YRI', 'CEU', 'CHB']
sample sizes: [10 10 10]
FST:
  YRI, CEU: 0.189
  YRI, CHB: 0.205
  CEU, CHB: 0.147
```

It's that simple. We can also simulate data for a subset of the populations, while still accounting for migration with other non-sampled populations:

```
sampled_demes = ["YRI"]
sample_sizes = [40]

fs_yri = moments.Spectrum.from_demes(
    ooa_model, sampled_demes=sampled_demes, sample_sizes=sample_sizes
)

print("populations:", fs_yri.pop_ids)
print("sample sizes:", fs_yri.sample_sizes)
print("Tajima's D =", f"{fs_yri.Tajima_D():.3}")
```

```
populations: ['YRI']
sample sizes: [40]
Tajima's D = -0.338
```

8.2.1 Ancient samples

Or sample a combination of ancient and modern samples from a population:

```
sampled_demes = ["CEU", "CEU"]
sample_sizes = [10, 10]
# sample size of 10 from present and 10 from 20,000 years ago
sample_times = [0, 20000]

fs_ancient = moments.Spectrum.from_demes(
    ooa_model,
    sampled_demes=sampled_demes,
    sample_sizes=sample_sizes,
    sample_times=sample_times,
)

print("populations:", fs.pop_ids)
print("sample sizes:", fs.sample_sizes)
print("FST(current, ancient) =", f"{fs.Fst():.3}")
```

```
populations: ['CEU', 'CEU_sampled_20000_0']
sample sizes: [10 10]
FST(current, ancient) = 0.0912
```

Note the population IDs, which are appended with "_sampled_{at_time}" where "at_time" is the generation or year (depending on the time unit of the model), as a float with an underscore replacing the decimal (here, 20000.0 years ago).

8.2.2 Alternative samples specification

By specifying sampled demes, sample sizes, and sample times, we have a lot of flexibility over the sampling scheme. Samples can more simply be specified as a dictionary, with one key per sampled population and values specifying sample sizes. This dictionary is passed to the `from_demes` function using the `samples` keyword, and it cannot be used in conjunction with sample times. As such, samples are taken at the end time (most recent time) of each population.

```
samples = {"YRI": 10, "CEU": 20, "CHB": 30, "OOA": 10}
fs = moments.Spectrum.from_demes(ooa_model, samples=samples)
```

Here, samples from YRI, CEU, and CHB are taken from time zero, and the OOA sample is taken from just before its split into the CEU and CHB branches.

8.2.3 Linkage disequilibrium

We can similarly compute *LD statistics*. Here, we compute the set of multi-population Hill-Robertson statistics for the three contemporary populations (YRI, CEU, and CHB), for three different recombination rates, $\rho = 4Nr = 0, 1, 2$.

```
import moments.LD

sampled_demes = ["YRI", "CEU", "CHB"]
y = moments.LD.LDstats.from_demes(
    ooa_model, sampled_demes=sampled_demes, rho=[0, 1, 2]
)

print("sampled populations:", y.pop_ids)
```

```
sampled populations: ['YRI', 'CEU', 'CHB']
```

8.2.4 Selection and dominance in Demes.SFS

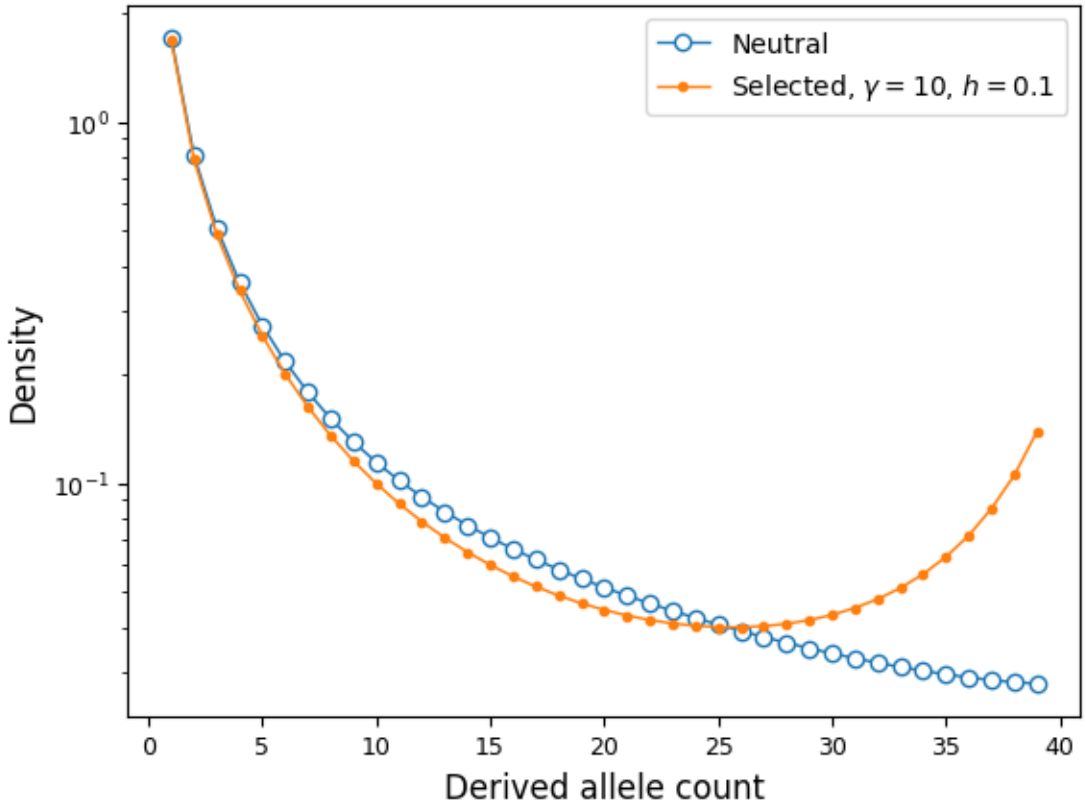
Moments can compute the SFS under selection and dominance. The `demes` model format currently lets us specify a single selection and dominance coefficient for each population in the model, or we can set different selection parameters in each populations.

The most simple scenario is to specify a single selection and dominance parameter that applied to all populations in the demographic model. In this case, we can pass `gamma` and/or `h` as scalar values to the function `moments.Spectrum.from_demes()`:

```
sampled_demes = ["YRI"]
sample_sizes = [40]
gamma = 10
h = 0.1

fs_yri_sel = moments.Spectrum.from_demes(
     ooa_model,
     sampled_demes=sampled_demes,
     sample_sizes=sample_sizes,
     gamma=gamma,
     h=h
)
```

We can compare the neutral and selected spectra:

```
# compare to neutral SFS for YRI
fig = plt.figure()
ax = plt.subplot(111)
ax.semilogy(fs_yri, "-o", ms=6, lw=1, mfc="w", label="Neutral");
ax.semilogy(fs_yri_sel, "-o", ms=3, lw=1,
    label=f"Selected, $\gamma={gamma}$, $h={h}$");
ax.set_ylabel("Density");
ax.set_xlabel("Derived allele count");
ax.legend();
```



We can gain more fine-grained control over variable selection and dominance in different populations by specifying `gamma` and `h` as dictionaries mapping population names to the coefficients. There can be as many different coefficient values as there are different demes in the demographic model. However, if a population is missing from the dictionary, it is assigned the default selection or dominance coefficient. In most cases the default values are $\gamma = 0$ and $h = 1/2$, but these can be changed by specifying a `_default` value in the selection and dominance dictionaries.

For example:

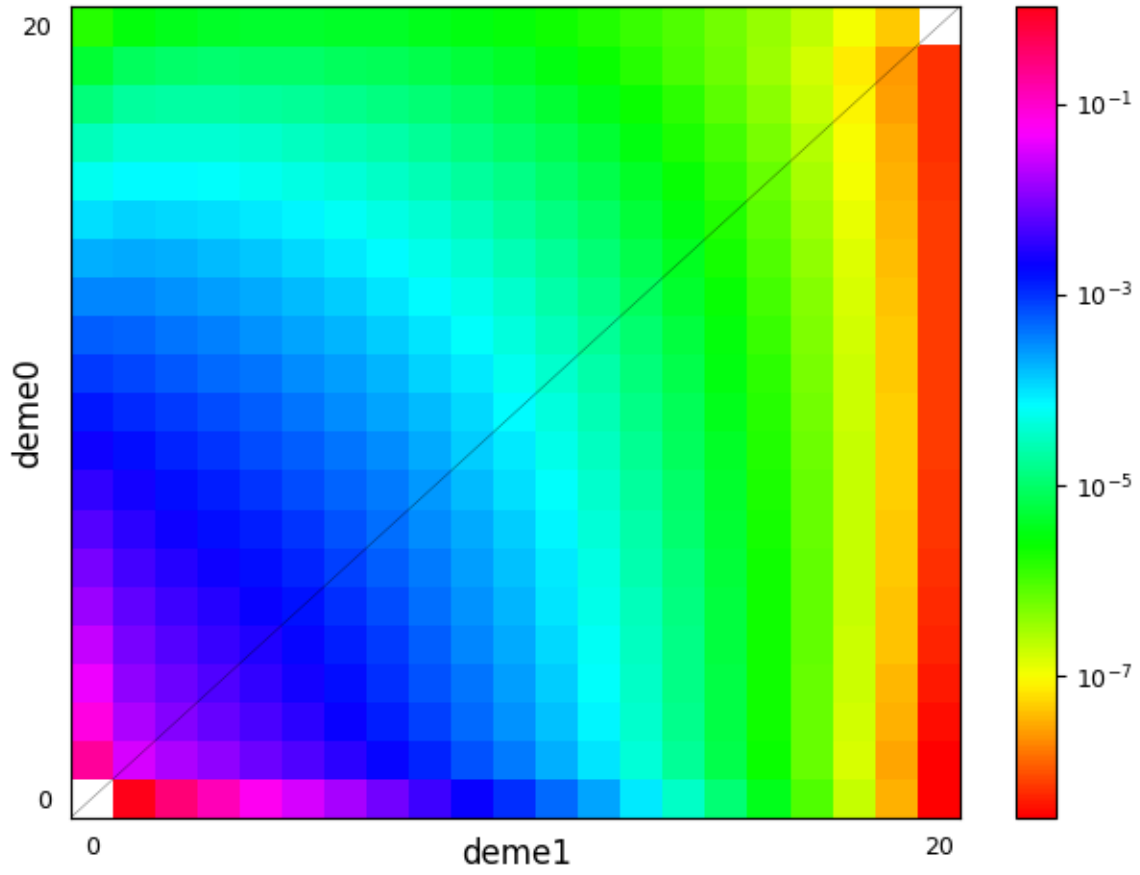
```
g = demes.load("data/im-parsing-example.yaml")
print(g)

gamma = {"anc": -10, "deme0": -10, "deme1": 5}
h = {"anc": 0.3, "deme0": 0.3, "deme1": 0.7}

fs = moments.Spectrum.from_demes(
    g,
```

(continues on next page)

(continued from previous page)

```
    sampled_demes=["deme0", "deme1"],
    sample_sizes=[20, 20],
    gamma=gamma,
    h=h
)

moments.Plotting.plot_single_2d_sfs(fs)
```

```
description: A simple isolation-with-migration model
time_units: generations
generation_time: 1
demes:
- name: anc
  epochs:
  - {end_time: 1500, start_size: 10000}
- name: deme0
  ancestors: [anc]
  epochs:
  - {end_time: 0, start_size: 2000}
- name: deme1
  ancestors: [anc]
  epochs:
  - {end_time: 0, start_size: 20000}
migrations:
- demes: [deme0, deme1]
  rate: 0.0001
```



In the case that a demographic model has many populations but only a small subset have differing selection or dominance strengths, we can assign a default value different from $s = 0$ or $h = 1/2$. This is done by including a *_default* key in the dictionary (note the leading underscore, to minimize the chance that the default key conflicts with a named population in the demographic model). Taking the example above:

```
gamma = {"_default": -10, "deme1": 5}
h = {"_default": 0.3, "deme1": 0.7}

fs_defaults = moments.Spectrum.from_demes(
    g,
    sampled_demes=["deme0", "deme1"],
    sample_sizes=[20, 20],
    gamma=gamma,
    h=h
)

assert np.allclose(fs, fs_defaults)
```

8.3 Using Demes to infer demography

Above, we showed how to use moments and demes-based demographic models to compute expectations for *static* demographic models. That is, given a fixed demography we can compute expectations for the SFS or LD. We often want to optimize the parameters of a given demographic model to fit observations from data. The general idea is that we specify a parameterized model, compute the expected SFS under that model and its likelihood given the data, and then update the model parameters to improve the fit. Moments uses scipy's optimization functions to perform optimization.

To run the inference, we need three items: 1) the data (SFS) to be fit, 2) a parameterized demographic model, and 3) a way to tell the optimization function which parameters to fit and any constraints on those parameters. We'll assume you already have a data SFS with stored pop_ids. For example, the data could be a 3-dimensional SFS for the three sampled populations in the Out-of-Africa demographic model above, so that data.pop_ids = ["YRI", "CEU", "CHB"].

The second item is the demes-formatted demographic model, such as the model written above. In this model, the parameter values are the demographic event times, population sizes, and migration rates, and the YAML file specifies all fixed parameters and initial guesses for the parameters to be fit.

The third item is a separate YAML-formatted file that tells the optimization function the variable parameters that should be fit and any bounds and/or inequality constraints on the parameter values.

8.3.1 The options file

All parameters to be fit must be included under parameters in the option file. Any parameter that is not included here is assumed to be a fixed parameter, and it will remain the value given in the Demes graph. moments will read this YAML file into a dictionary using a YAML parser, so it needs to be valid and properly formatted YAML code.

The only required field in the "options" YAML is parameters. For each parameter to be fit, we must name that parameter, which can be any unique string, and we need to specify which values in the Demes graph correspond to that value (optionally, we can include a parameter description for our own sake). For example, to fit the bottleneck size in the Out-of-Africa model, our options file would look like:

```
parameters:
- name: N_B
  description: Bottleneck size for Eurasian populations
  values:
  - demes:
      OOA:
        epochs:
          0: start_size
  lower_bound: 100
  upper_bound: 100000
```

This specifies that the start size of the first (and only) epoch of the OOA deme in the Demes graph should be fit. We have also specified that the fit for this parameter should be bounded between 100 and 100,000.

The same parameter can affect multiple values in the Demes graph. For example, the size of the African population in the Out-of-Africa model is applied to both the AMH and the YRI demes. This simply requires adding additional keys in the values entry:

```
parameters:
- name: N_A
  description: Expansion size
    values:
    - demes:
```

(continues on next page)

(continued from previous page)

```
        AMH:
          epochs:
            0: start_size
        YRI:
          epochs:
            0: start_size
  lower_bound: 100
  upper_bound: 100000
```

Migration rates can be specified to be fit as well. Note that the index of the migration is given, pointing to the migrations in the order they are specified in the demes file.

```
parameters:
- name: m_Af_Eu
  description: Symmetric migration rate between Afr and Eur populations
  upper_bound: 1e-3
  values:
  - migrations:
      1: rate
```

Note here that we have specified the upper bound to be `1e-3` (the units of the migration rate are parental migrant probabilities, typical in population genetics models). For any parameter, we can set the lower bound and upper bound as shown here. If they are not given, the lower bound defaults to 0 and the upper bound defaults to infinity.

Finally, we can also specify constraints on parameters. For example, if some event necessarily occurs before another, we should add that relationship to the list of constraints.

```
parameters:
- name: TA
  description: Time before present of ancestral expansion
  values:
  - demes:
      ancestral:
        epochs:
          0: end_time
- name: TB
  description: Time of YRI-OOA split
  values:
  - demes:
      AMH:
        epochs:
          0: end_time
- name: TF
  description: Time of CEU-CHB split
  values:
  - demes:
      OOA:
        epochs:
          0: end_time
constraints:
- params: [TA, TB]
  constraint: greater_than
- params: [TB, TF]
```

(continues on next page)

(continued from previous page)

```
  constraint: greater_than
```

This specifies each of the event timings in the OOA model to be fit, and the constraints say that `TA` must be greater than `TB`, and `TB` must be greater than `TF`.

8.3.2 The inference function

To run optimization using the Demes modeul, we call `moments.Demes.Inference.optimize`. The first three required inputs to `optimize` are the Demes input graph, the parameter options, and the data, in that order.

Additional options can be passed to the optimization function using keyword arguments in the `moments.Demes.Inference.optimize` function. These include:

- `maxiter`: Maximum number of iterations to run optimization. Defaults to 1,000.
- `perturb`: Defaults to 0 (no perturbation of initial parameters). If greater than zero, it perturbs the initial parameters by up to `perturb`-fold. So if `perturb` is 1, initial parameters are randomly chosen from $[1/2 \times p_0, 2 \times p_0]$. Larger values result in stronger perturbation of initial guesses.
- `verbose`: Defaults to 0. If greater than zero, it prints an update to the specified `output_stream` (which defaults to `sys.stdout`) every `verbose` iterations.
- `uL`: Defaults to None. If given, this is the product of the per-base mutation rate and the length of the callable genome used to compile the data SFS. If we don't give this scaled mutation rate, we optimize with theta as a free parameter. Otherwise, we optimize with theta given by $\theta = 4 \times N_e \times uL$, and N_e is taken to be the size of the root/ancestral deme (for which the size can be a either be a fixed parameter or a parameter to be fit!).
- `log`: Defaults to True. If True, optimize the log of the parameters.
- `method`: The optimization method to use, currently with the options "fmin" (Nelder-Mead), "powell", or "lbfgsb". Defaults to "fmin".
- `fit_ancestral_misid`: Defaults to False, and cannot be used with a folded SFS. For an unfolded SFS, the ancestral state may be misidentified, resulting in a distortion of the SFS. We can account for that distortion by fitting a parameter that accounts for some fraction of mis-labeled ancestral states.
- `misid_guess`: Used with `fit_ancestral_misid`, as the initial ancestral misidentification parameter guess. Defaults to 0.02.
- `output_stream`: Defaults to sys.stdout.
- `output`: Defaults to None, in which case the result is printed to the `output_stream`. If given, write the optimized Demes graph in YAML format to the given path/filename.
- `overwrite`: Defaults to False. If True, we overwrite any file with the path/filename given by `output`.

8.4 Single-population inference example

To demonstrate, we'll fit a simple single-population demographic model to the synonymous variant SFS in the Mende (MSL) from the Thousand Genomes data. The data for this population is stored in the docs/data directory. We previous parsed all coding variation and used a mutation model to estimate $u \times L$.

We can either fold the frequency spectrum, which is useful when we do not know the ancestral states of mutations. Alternatively, we can fit with the unfolded spectrum, and if we suspect that some proportion of SNPs have their ancestral state misidentified, we can additionally fit a parameter that corrects for this uncertainty. We'll take the second approach here, and fit the unfolded spectrum.

```
import moments
import pickle

all_data = pickle.load(open("./data/msl_data.bp", "rb"))
data = all_data["spectra"]["syn"]
data.pop_ids = ["MSL"]
uL = all_data["rates"]["syn"]
print("scaled mutation rate (u_syn * L):", uL)

# project down to a smaller sample size, for illustration purposes
data = data.project([30])
```

```
scaled mutation rate (u_syn * L): 0.14419746897690008
```

We'll fit a demographic model that includes an ancient expansion and a more recent exponential growth. This initial model is stored in the docs/data directory as well.

The YAML specification of this model is

```
description: A single-population model to be fit to the MSL data. Initial guesses
  are given as parameters in the model.
time_units: years
generation_time: 29
demes:
- name: MSL
  epochs:
  - end_time: 350000
    start_size: 10000
  - end_time: 20000
    start_size: 25000
  - end_time: 0
    end_size: 60000
```

And we can specify that we want to fit the times of the size changes, and all population sizes. (Note that if we did not have an estimate for the mutation rate, we would not fit the ancestral size.)

```
parameters:
- name: T1
  description: Time before present of ancestral expansion
  values:
  - demes:
      MSL:
        epochs:
          0: end_time
- name: T2
  description: Time before present of start of exponential growth.
  values:
  - demes:
      MSL:
        epochs:
          1: end_time
- name: Ne
  description: Effective (ancestral/root) size
```

(continues on next page)

(continued from previous page)

```
  values:
  - demes:
      MSL:
        epochs:
          0: start_size
- name: NA
  description: Ancestral expansion size
  values:
  - demes:
      MSL:
        epochs:
          1: start_size
- name: NF
  description: Final population size
  values:
  - demes:
      MSL:
        epochs:
          2: end_size
constraints:
- params: [T1, T2]
  constraint: greater_than
```

```
deme_graph = "./data/msl_initial_model.yaml"
options = "./data/msl_options.yaml"
```

And now we can run the inference:

```
output = "./data/msl_best_fit_model.yaml"
ret = moments.Demes.Inference.optimize(
    deme_graph,
    options,
    data,
    uL=uL,
    fit_ancestral_misid=True,
    misid_guess=0.01,
    method="lbfgsb",
    output=output,
    overwrite=True
)
param_names, opt_params, LL = ret
print("Log-likelihood:", -LL)
print("Best fit parameters")
for n, p in zip(param_names, opt_params):
    print(f"{n}\t{p:.3}")
```

```
Log-likelihood: -126.08823318048917
Best fit parameters
T1      4.38e+05
T2      2.2e+04
Ne      1.09e+04
NA      2.5e+04
```

(continues on next page)

(continued from previous page)

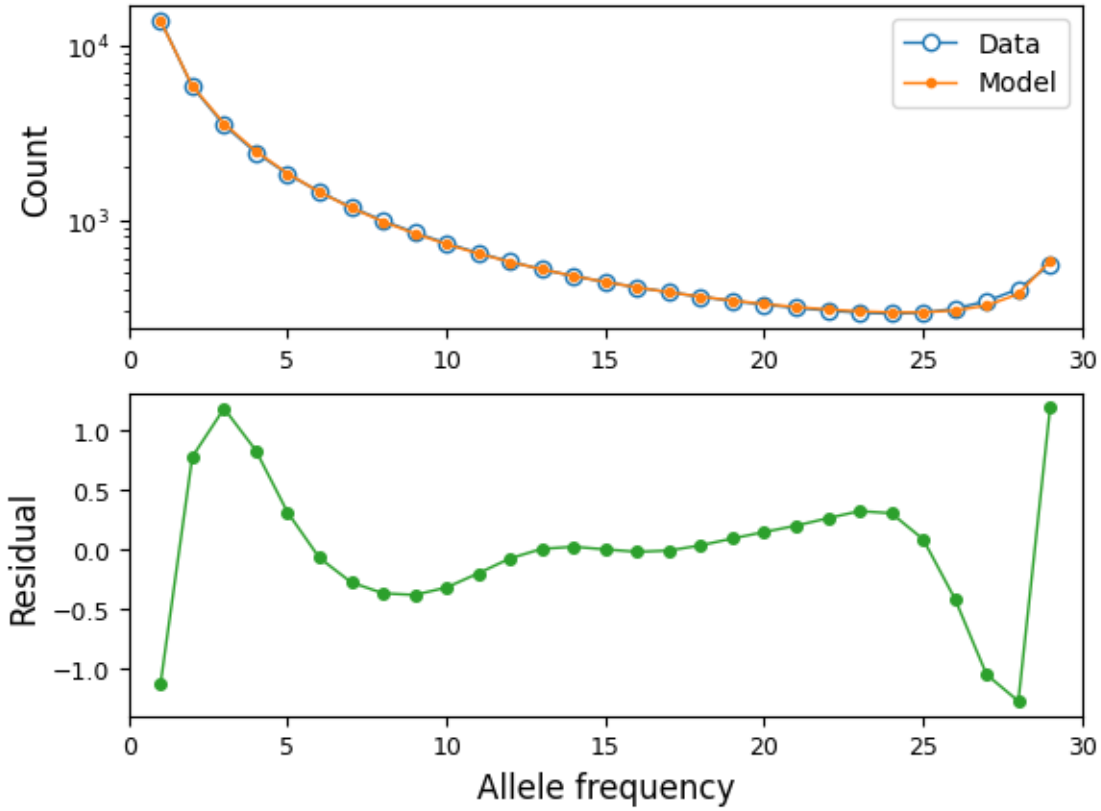
```
NF          6.6e+04
p_misid         0.026
```

Printed above are the best fit parameters for this model, including the ancestral misidentification rate for synonymous variants in the Mende sample. Parameters in this fit are scaled by our estimate of the total mutation rate of synonymous variants (uL), which allows us to infer the ancestral N_e. Below, we plot the results and then compute confidence intervals for this fit.

8.5 Plotting the results

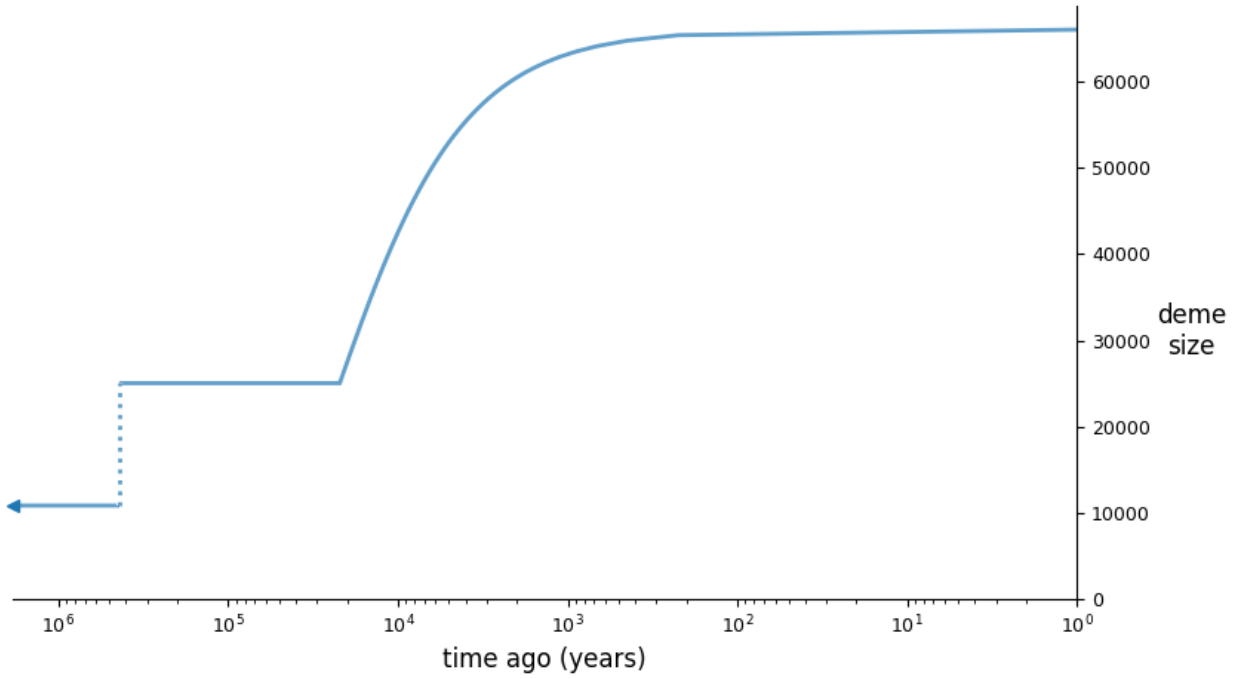
We can see how well our best fit model fits the data, using `moments` plotting features:

```
fs = moments.Spectrum.from_demes(output, samples={"MSL": data.sample_sizes})
fs = moments.Misc.flip_ancestral_misid(fs, opt_params[-1])
moments.Plotting.plot_1d_comp_multinom(fs, data)
```



And we can illustrate the best fit model using demesdraw:

```
import demes, demesdraw
opt_model = demes.load(output)
demesdraw.size_history(opt_model, invert_x=True, log_time=True);
```



8.6 Computing confidence intervals

Using the output YAML from `moments.Demes.Inference.optimize()`, we compute confidence intervals using `moments.Demes.Inference.uncerts()`. This function takes the output Demes graph from the optimization, the same parameter options file, and the same data used in inference. These need to be consistent between the optimization and uncertainty computation. If we specified the mutation rate or inferred an ancestral misidentification parameter, those must also be provided.

The additional options to `uncerts()` are

- `bootstraps`: Defaults to None, in which case we use the FIM approach.
- `uL`: The scaled mutation rate, if used in the optimization. (See above for details.)
- `log`: Defaults to False. If True, we assume a log-normal distribution of parameters. Returned values are then the standard deviations of the *logs* of the parameter values, which can be interpreted as relative parameter uncertainties.
- `eps`: The relative step size to use when numerically computing derivatives to estimate the curvature of the likelihood function at the inferred best-fit parameters.
- `method`: Defaults to "FIM", which uses the Fisher information matrix. We can also use the Godambe information matrix, which uses bootstrap replicates to account for non-independence between linked SNPs. This uses methods developed by Alec Coffman in Ryan Gutenkunst's group, described in [Coffman2016].
- `fit_ancestral_misid`: If the ancestral misid was fit, this should be set to True.
- `misid_fit`: The fit misidentification parameter, if it was fit.
- `output_stream`: Defaults to sys.stdout.

In our example using the Mende data above, we'll use the `FIM` method compute confidence intervals:

```
std_err = moments.Demes.Inference.uncerts(
    output,
    options,
    data,
    uL=uL,
    fit_ancestral_misid=True,
    misid_fit=opt_params[-1],
)

print("95% CIs")
print("param\t\t2.5%\t\t97.5%")
for n, p, e in zip(param_names, opt_params, std_err):
    print(f"{n}\t{p - 1.96 * e:-12g}\t{p + 1.96 * e:-13g}")
```

```
95% CIs
param               2.5%               97.5%
T1               378253              497820
T2              10528.9             33528.2
Ne              10387.3             11316.9
NA              23521.2             26539.5
NF              41670.7             90313.3
p_misid           0.0228469           0.0291068
```

To compute standard errors that account for non-independence between SNPs, we would use `method="GIM"` and include a list of bootstrap replicate spectra that we pass to `bootstraps`.

8.7 Two-population inference and uncertainty example

Here, we'll simulate a demographic model using `msprime`. In this example, we'll simulate many regions of varying length and mutation rates, from which we compute `uL` and estimate confidences using the `GIM` method, which requires bootstrapped datasets of the SFS and associated scaled mutation rates.

First, we'll simulate data under this two-population model:

```
g = demes.load("./data/two-deme-example.yaml")
print(g)
```

```
time_units: generations
generation_time: 1
demes:
- name: anc
  epochs:
  - {end_time: 2000, start_size: 8500}
- name: A
  ancestors: [anc]
  epochs:
  - {end_time: 0, start_size: 700, end_size: 11000}
- name: B
  ancestors: [anc]
  epochs:
  - {end_time: 0, start_size: 17500}
```

(continues on next page)

(continued from previous page)

```
migrations:
- demes: [A, B]
  rate: 0.0015
```

```
import msprime

demog = msprime.Demography.from_demes(g)

num_regions = 200
# Lengths between 75 and 125 kb
Ls = np.random.randint(75000, 125000, 200)
# Mutation rates between 1e-8 and 2e-8
us = 1e-8 + 1e-8 * np.random.rand(200)

# Total mutation rate
uL = np.sum(us * Ls)

# Simulate and store allele frequency data (summed and by region)
ns = [20, 20]
region_data = {}
data = moments.Spectrum(np.zeros((ns[0] + 1, ns[1] + 1)))
data.pop_ids = ["A", "B"]
# sample_sets are required to get the SFS from the tree sequences
sample_sets = (range(20), range(20, 40))

for i, (u, L) in enumerate(zip(us, Ls)):
    ts = msprime.sim_ancestry(
        {"A": ns[0] // 2, "B": ns[1] // 2},
        demography=demog,
        recombination_rate=1e-8,
        sequence_length=L,
    )
    ts = msprime.sim_mutations(ts, rate=u)
    SFS = ts.allele_frequency_spectrum(
        sample_sets=sample_sets, span_normalise=False, polarised=True)
    region_data[i] = {"uL": u * L, "SFS": SFS}
    data += SFS

print("Simulated data. FST =", data.Fst())
```

```
Simulated data. FST = 0.03488586466502664
```

With this simulated data, we can now re-infer the model, using the following options:

```
parameters:
- name: T
  description: Ancestral split time
  values:
  - demes:
      anc:
        epochs:
```

(continues on next page)

(continued from previous page)

```
          0: end_time
- name: Ne
  description: Ancestral effective population size
  values:
  - demes:
      anc:
        epochs:
          0: start_size
- name: NA0
  description: Initial population size of A
  values:
  - demes:
      A:
        epochs:
          0: start_size
- name: NA
  description: Final population size of A
  values:
  - demes:
      A:
        epochs:
          0: end_size
- name: NB
  description: B population size
  values:
  - demes:
      B:
        epochs:
          0: start_size
- name: M
  description: migration rate between A and B
  values:
  - migrations:
      0: rate
  upper_bound: 1
```

```
deme_graph = "./data/two-deme-example.yaml"
options = "./data/two-deme-example-options.yaml"
output = "./data/two-deme-example-best-fit.yaml"

ret = moments.Demes.Inference.optimize(
    deme_graph,
    options,
    data,
    uL=uL,
    perturb=1,
    output=output,
    overwrite=True
)
```

Printing the results of this inference run:

```
param_names, opt_params, LL = ret
print("Log-likelihood:", -LL)
print("Best fit parameters")
for n, p in zip(param_names, opt_params):
    print(f"{n}\t{p:.3}")
```

```
Log-likelihood: -1296.725088604149
Best fit parameters
T        1.52e+03
Ne        8.61e+03
NA0        5.74e+02
NA        2.05e+04
NB        1.46e+04
M        0.00115
```

To compute confidence intervals using the Godambe method, we need generate bootstrap replicates of the data (and scaled mutation rate, if specified in the optimization).

```
bootstraps = []
bootstraps_uL = []
for _ in range(len(region_data)):
    choices = np.random.choice(range(200), 200, replace=True)
    bootstraps.append(
        moments.Spectrum(sum([region_data[c]["SFS"] for c in choices])))
    bootstraps_uL.append(sum([region_data[c]["uL"] for c in choices]))
```

Computing the uncertainties using `GIM` requires passing the bootstrapped data:

```
std_err = moments.Demes.Inference.uncerts(
    output,
    options,
    data,
    bootstraps=bootstraps,
    uL=uL,
    bootstraps_uL=bootstraps_uL,
    method="GIM",
)
```

```
print("Standard errors:")
print("param\t\topt\t\tstderr")
for n, p, e in zip(param_names, opt_params, std_err):
    print(f"{n}\t{p:-11g}\t{e:-14g}")
```

```
Standard errors:
param                opt                stderr
T             1515.2               183.786
Ne            8607.87               179.118
NA0            573.705               84.0684
NA              20466                5675.2
NB            14610.3               1994.22
M          0.00115135           0.000150782
```

8.8 References

CHAPTER

NINE

TWO-LOCUS FREQUENCY SPECTRUM

See *Selection at two loci* for introduction and examples to the Two-Locus extension.

9.1 API

The `TLSpectrum` class handles all manipulations of the two-locus frequency spectrum:

class moments.TwoLocus.**TLSpectrum**(*data*, *mask=False*, *mask_infeasible=True*, *mask_fixed=False*, *data_folded=None*, *check_folding=True*, *dtype=<class 'float'>*, *copy=True*, *fill_value=nan*, *keep_mask=True*, *shrink=True*)

Represents a two locus frequency spectrum.

Parameters

- **data** (*array*) – The frequency spectrum data, which has shape (n+1)-by-(n+1)-by-(n+1) where n is the sample size.
- **mask** (*array*) – An optional array of the same size as data. 'True' entries in this array are masked in the TLSpectrum.
- **mask_infeasible** (*bool*) – If True, mask all bins for frequencies that cannot occur, e.g. i + j > n. Defaults to True.
- **mask_fixed** (*bool*) – If True, mask the fixed bins. Defaults to True.
- **data_folded** (*bool*) – If True, it is assumed that the input data is folded for the major and minor derived alleles
- **check_folding** (*bool*) – If True and data_folded=True, the data and mask will be checked to ensure they are consistent.

D(*proj=True*, *nA=None*, *nB=None*)

Return the expectation of D from the spectrum.

Parameters

- **proj** – If True, use the unbiased estimator from downsampling. If False, use naive maximum likelihood estimates for frequency.
- **nA** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the left locus.
- **nB** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the right locus.

D2(*proj=True*, *nA=None*, *nB=None*)

Return the expectation of D^2 from the spectrum.

Parameters

- **proj** – If True, use the unbiased estimator from downsampling. If False, use naive maximum likelihood estimates for frequency.
- **nA** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the left locus.
- **nB** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the right locus.

Dz(*proj=True*, *nA=None*, *nB=None*)

Compute the expectation of $Dz = D(1-2p)(1-2q)$ from the spectrum.

Parameters

- **proj** – If True, use the unbiased estimator from downsampling. If False, use naive maximum likelihood estimates for frequency.
- **nA** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the left locus.
- **nB** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the right locus.

S(*nA=None*, *nB=None*)

Return the sum of probabilities over all variable two-locus entries in the spectrum.

ancestral_misid(*p*)

Return a new SFS with a given ancestral misidentification, p.

Parameters

p – The rate of ancestral state misidentification.

fold()

Fold the two-locus spectrum by minor allele frequencies.

static from_file(*fid*, *mask_infeasible=True*, *return_comments=False*)

Read frequency spectrum from file.

Parameters

- **fid** (*str*) – String with file name to read from or an open file object.
- **mask_infeasible** (*bool*) – If True, mask the infeasible entries in the two locus spectrum.
- **return_comments** (*bool*) – If true, the return value is (fs, comments), where comments is a list of strings containing the comments from the file.

integrate(*nu*, *tf*, *dt=0.01*, *rho=None*, *gamma=None*, *sel_params=None*, *sel_params_general=None*, *theta=1.0*, *finite_genome=False*, *u=None*, *v=None*, *alternate_fg=None*, *clustered_mutations=False*)

Simulate the two-locus haplotype frequency spectrum forward in time. This integration scheme takes advantage of scipy's sparse methods.

When using the reversible mutation model (with *finite_genome* = True), we are limited to selection at only one locus (the left locus), and selection is additive. When using the default ISM, additive selection is allowed at both loci, and we use *sel_params*, which specifies [sAB, sA, and sB] in that order. Note that while this selection model is additive within loci, it allows for epistasis between loci if sAB != sA + sB.

Parameters

- **nu** – Population effective size as positive value or callable function.
- **tf** (*float*) – The integration time in genetics units.
- **dt_fac** (*float*) – The time step for integration.
- **rho** (*float*) – The population-size scaled recombination rate 4*Ne*r.
- **gamma** (*float*) – The population-size scaled selection coefficient 2*Ne*s.
- **sel_params** (*list*) – A list of selection parameters. See docstrings in Numerics. Selection parameters will be deprecated when we clean up the numerics and integration.
- **sel_params_general** (*list*) – To be filled. ## TODO!!
- **theta** (*float*) – Population size scale mutation parameter.
- **finite_genome** (*bool*) – Defaults to False, in which case we use the infinite sites model. Otherwise, we use a reversible mutation model, and must specify u and v.
- **u** (*float*) – The mutation rate at the left locus in the finite genome model.
- **v** (*float*) – The mutation rate at the right locus in the finite genome model.
- **alternate_fg** (*bool*) – If True, use the alternative finite genome model. This parameter will be deprecated when we clean up the numerics and integration.

left()

The marginal allele frequency spectrum at the left locus.

mask_fixed()

Mask all infeasible entries, as well as any where both sites are not segregating.

mask_infeasible()

Mask any infeasible entries.

pi2(*proj=True*, *nA=None*, *nB=None*)

Return the expectation of $\pi_2 = p(1-p)q(1-q)$ from the spectrum.

Parameters

- **proj** – If True, use the unbiased estimator from downsampling. If False, use naive maximum likelihood estimates for frequency.
- **nA** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the left locus.
- **nB** – If None, the average is computed over all frequencies. If given, condition on the given allele count for the right locus.

project(*ns*, *finite_genome=False*, *cache=True*)

Project to smaller sample size.

param int ns: Sample size for new spectrum. param bool finite_genome: If we also track proportions in fixed bins.

right()

The marginal AFS at the right locus.

to_file(*fid*, *precision=16*, *comment_lines=[]*, *foldmaskinfo=True*)

Write frequency spectrum to file.

Parameters

- **fid** (*str*) – String with file name to write to or an open file object.
- **precision** (*int*) – Precision with which to write out entries of the SFS. (They are formated via %.<p>g, where <p> is the precision.)
- **comment_lines** (*list*) – List of strings to be used as comment lines in the header of the output file.
- **foldmaskinfo** (*bool*) – If False, folding and mask and population label information will not be saved.

unfold()

Remove folding from the spectrum.

The Demographics module contains some standard demographic models. This is a good place to look for some inspiration to create your own two-locus models as well.

moments.TwoLocus.Demographics.**bottlegrowth**(*params*, *ns*, *rho=None*, *theta=1.0*, *gamma=None*, *sel_params=None*)

A bottleneck followed by exponential growth. The population changes size to nuB T generations ago, and then has exponential size change to final size nuF. Time is in units of 2Ne generations, and sizes are relative to the ancestral Ne.

Parameters

- **params** – Given as [nuB, nuF, T].
- **ns** – The sample size.
- **rho** – The population size scaled selection coefficient, 4*Ne*r.
- **theta** – The mutation rate at each locus, typically left as 1.
- **gamma** – Only used for additive selection at the A/a locus.
- **sel_params** – Additive selection coefficients for haplotypes AB, Ab, and aB, so that sel_params = [sAB, sA, sB]. If sAB = sA + sB, this is a model with no epistasis.

moments.TwoLocus.Demographics.**equilibrium**(*ns*, *rho=None*, *theta=1.0*, *gamma=None*, *sel_params=None*, *sel_params_general=None*, *cache=False*)

Compute or load the equilibrium two locus frequency spectrum. If the cached spectrum does not exist, create the equilibrium spectrum and cache in the cache path.

Parameters

- **ns** – The sample size.
- **rho** – The population size scaled selection coefficient, 4*Ne*r.
- **theta** – The mutation rate at each locus, typically left as 1.
- **gamma** – Only used for additive selection at the A/a locus.
- **sel_params** – Additive selection coefficients for haplotypes AB, Ab, and aB, so that sel_params = [sAB, sA, sB]. If sAB = sA + sB, this is a model with no epistasis.
- **sel_params_general** – General selection parameters for diploids. In the order (s_AB_AB, s_AB_Ab, s_AB_aB, s_AB_ab, s_Ab_Ab, s_Ab_aB, s_Ab_ab, s_aB_aB, s_aB_ab)
- **cache** – If True, save the frequency spectrum in the cache for future use. If False, don't save the spectrum.

moments.TwoLocus.Demographics.**growth**(*params*, *ns*, *rho=None*, *theta=1.0*, *gamma=None*, *sel_params=None*)

An expnential growth model, that begins growth at time T ago, in units of 2Ne generations. The final size is given by nu, which is the relative size to the ancestral Ne.

Parameters

- **params** – Given as [nu, T].
- **ns** – The sample size.
- **rho** – The population size scaled selection coefficient, 4*Ne*r.
- **theta** – The mutation rate at each locus, typically left as 1.
- **gamma** – Only used for additive selection at the A/a locus.
- **sel_params** – Additive selection coefficients for haplotypes AB, Ab, and aB, so that sel_params = [sAB, sA, sB]. If sAB = sA + sB, this is a model with no epistasis.

moments.TwoLocus.Demographics.**set_cache_path**(*path='~/.moments/TwoLocus_cache'*)

Set directory in which demographic equilibrium phi spectra will be cached.

The collection of cached spectra can get large, so it may be helpful to store them outside the user's home directory.

moments.TwoLocus.Demographics.**three_epoch**(*params*, *ns*, *rho=None*, *theta=1.0*, *gamma=None*, *sel_params=None*)

A three-epoch model, with relative size changes nu1 that lasts for time T1, followed by a relative size change to nu2 that last for time T2. Times are in units of 2Ne generations, and sizes are relative to the ancestral Ne.

Parameters

- **params** – Given as [nu1, nu2, T1, T2].
- **ns** – The sample size.
- **rho** – The population size scaled selection coefficient, 4*Ne*r.
- **theta** – The mutation rate at each locus, typically left as 1.
- **gamma** – Only used for additive selection at the A/a locus.
- **sel_params** – Additive selection coefficients for haplotypes AB, Ab, and aB, so that sel_params = [sAB, sA, sB]. If sAB = sA + sB, this is a model with no epistasis.

moments.TwoLocus.Demographics.**two_epoch**(*params*, *ns*, *rho=None*, *theta=1.0*, *gamma=None*, *sel_params=None*)

A two-epoch model, with relative size change nu, time T in the past. T is given in units of 2Ne generations. Note that a relative size of 1 implies no size change.

Parameters

- **params** – Given as [nu, T].
- **ns** – The sample size.
- **rho** – The population size scaled selection coefficient, 4*Ne*r.
- **theta** – The mutation rate at each locus, typically left as 1.
- **gamma** – Only used for additive selection at the A/a locus.
- **sel_params** – Additive selection coefficients for haplotypes AB, Ab, and aB, so that sel_params = [sAB, sA, sB]. If sAB = sA + sB, this is a model with no epistasis.

CHAPTER

TEN

TRIALLELE FREQUENCY SPECTRUM

10.1 API

class moments.Triallele.**TriSpectrum**(*data*, *mask=False*, *finite_genome=False*, *mask_infeasible=True*, *mask_fixed=True*, *data_folded_major=None*, *check_folding_major=True*, *data_folded_ancestral=None*, *check_folding_ancestral=True*, *dtype=<class 'float'>*, *copy=True*, *fill_value=nan*, *keep_mask=True*, *shrink=True*)

Represents a triallelic frequency spectrum.

The triallelic spectrum is represented as a square numpy masked array in which the (i, j)-th element stores the count or density of loci in which there are i copies of the first derived allele and j copies of the second derived allele.

Parameters

- **data** (*array*) – The frequency spectrum data of size (n+1)-by-(n+1) where n is the sample size.
- **mask** (*array*) – An optional array of the same size as data. 'True' entries in this array are masked in the TriSpectrum. These represent missing data categories, or invalid entries in the array
- **mask_infeasible** (*bool*) – If True, mask all bins for frequencies that cannot occur, e.g. i + j > n. Defaults to True.
- **mask_fixed** (*bool*) – If True, mask the fixed bins. Defaults to True.
- **data_folded_major** (*bool*) – If True, it is assumed that the input data is folded for the major and minor derived alleles.
- **data_folded_ancestral** (*bool*) – If True, it is assumed that the input data is folded to account for uncertainty in the ancestral state. Note that if True, data_folded_major must also be True.
- **check_folding_major** (*bool*) – If True and data_folded_ancestral=True, the data and mask will be checked to ensure they are consistent.
- **check_folding_ancestral** (*bool*) – If True and data_folded_ancestral=True, the data and mask will be checked to ensure they are consistent.

S()

Number of sites in the unmasked spectrum.

fold_ancestral()

Fold the spectrum based on the ancestral state

`fold_major()`

Fold the spectrum based on the major allele(s).

`static from_file(fid, mask_infeasible=True, return_comments=False)`

Read frequency spectrum from file.

See to_file method for details on the file format.

Parameters

- **fid** (*str*) – String with file name to read from or an open file object.
- **mask_infeasible** (*bool*) – If True, mask the infeasible entries in the triallelic spectrum.
- **return_comments** (*bool*) – If true, the return value is (fs, comments), where comments is a list of strings containing the comments from the file.

`integrate(nu, tf, dt=0.001, gammas=None, theta=1.0)`

Method to simulate the triallelic fs forward in time. This integration scheme takes advantage of scipy's sparse methods.

Parameters

- **nu** – The population effective size as positive value or callable function.
- **tf** (*float*) – The integration time in genetics units.
- **dt_fac** (*float*) – time step for integration
- **gammas** (*list*) – Population size scaled selection coefficients [sAA, sA0, sBB, sB0, sAB]. Here, 0 represents that ancestral allele, so we can implement dominance by picking the relationship between, e.g., sAA, sA0, sAB, and sA0.
- **theta** (*float*) – Population size scale mutation parameter, assuming equal mutation rates to both derived alleles.

`log()`

Return the natural logarithm of the entries of the frequency spectrum.

Only necessary because numpy.ma.log now fails to propagate extra attributes after numpy 1.10.

`mask_fixed()`

Mask entries that are not triallelic.

`mask_infeasible()`

Mask any infeasible entries.

`pi()`

Estimated expected number of pairwise differences between two samples from the population at loci that are triallelic

`project(ns, finite_genome=False)`

Project to smaller sample size.

Parameters

ns (*int*) – Sample size for new spectrum.

`to_file(fid, precision=16, comment_lines=[], foldmaskinfo=True)`

Write frequency spectrum to file.

The file format is:

- # Any number of comment lines beginning with a '#'

- A single line containing the sample size. On the *same line*, the string 'folded_major' or 'unfolded_major' denoting the folding status of the array. And on the *same line*, the string 'folded_ancestral' or 'unfolded_ancestral' denoting the folding status of the array.
- A single line giving the array elements. The order of elements is e.g.: fs[0, 0] fs[0, 1] fs[0, 2] … fs[1, 0] fs[1, 1] …
- A single line giving the elements of the mask in the same order as the data line. '1' indicates masked, '0' indicates unmasked.

Parameters

- **fid** (*str*) – String with file name to write to or an open file object.
- **precision** (*int*) – Precision with which to write out entries of the SFS. (They are formated via %.<p>g, where <p> is the precision.)
- **comment_lines** (*list*) – List of strings to be used as comment lines in the header of the output file.
- **foldmaskinfo** (*bool*) – If False, folding and mask and population label information will not be saved.

unfold()

Completely unfold the spectrum.

Returns a new TriSpectrum.

CHAPTER

ELEVEN

DEMOGRAPHY AND GENETIC DIVERSITY

Todo: This module has not been completed.

Intro - intuition about how demography is expected to affect summary statistics helps in hypothesizing historical scenarios to explain observed patterns of genetic diversity, or trouble-shooting poor fits of models to data. It's also important to understand how demographic parameters can be confounded and different evolutionary scenarios can give rise to similar patterns of genetic diversity.

11.1 Measures of genetic diversity

Many of the common single-site diversity statistics we are familiar with in population genetics are summaries of the SFS.

For single populations, diversity within a population is very often reported as the average heterozygosity (typically denoted π or H): the probability that two genome copies (i.e. samples) differ in state at a given locus. Suppose our SFS stores the distribution of allele frequencies over L loci for n samples. Then the expected or average π can be found by summing across allele frequency bins in the SFS and computing the probability that two randomly drawn copies carry different alleles for the given allele frequency:

$$\mathbf{E}[\pi] = \frac{1}{L}\sum_{i=1}^{n-1} 2\frac{i(n-i)}{n(n-1)}\text{SFS}(i)$$

Under the standard neutral model with steady-state demography, diversity is expected to be equal to the scaled mutation rate:

```
import moments

theta = 0.001 # the per-base scaled mutation rate, 4*Ne*u
n = 30 # the haploid sample size
fs = theta * moments.Demographics1D.snm([n])

print("Theta:", theta)
print("Diversity:", f"{fs.pi():0.4f}")
```

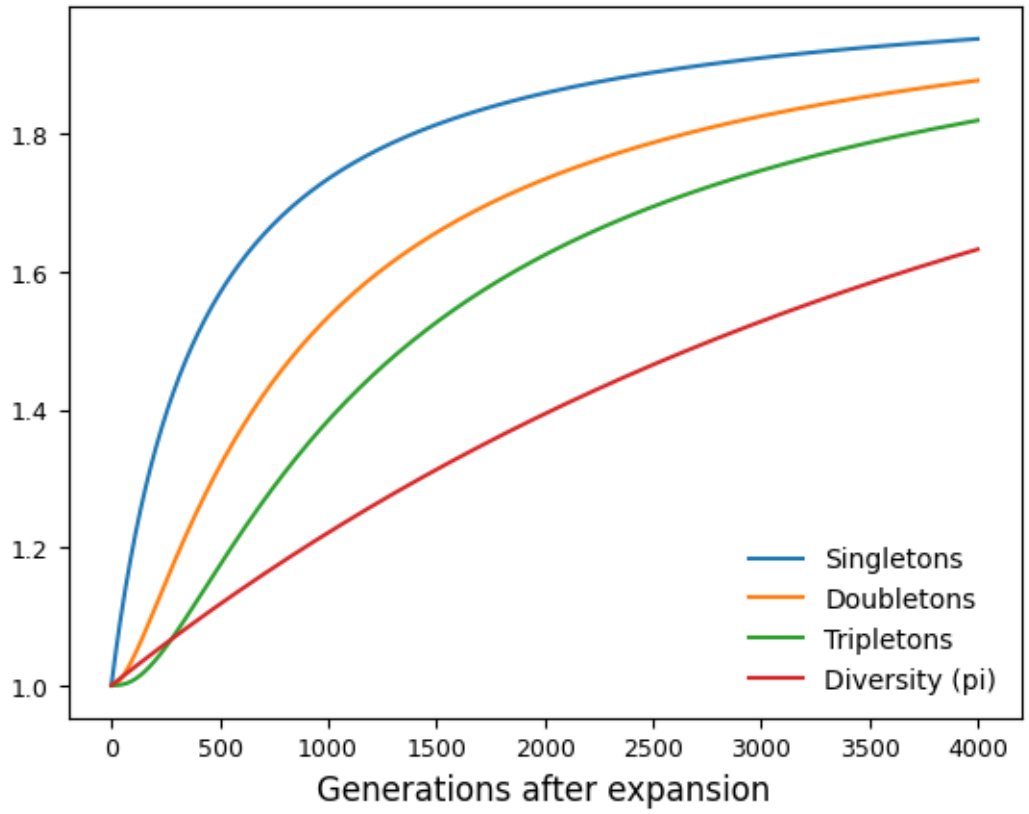
```
Theta: 0.001
Diversity: 0.0010
```

11.2 Single-population demography

Store values every x generations after instantaneouls double of size:

```
Ne = 1000

singletons = []
doubletons = []
tripletons = []
diversity = []

fs = moments.Demographics1D.snm([20])
singletons.append(fs[1])
doubletons.append(fs[2])
tripletons.append(fs[3])
diversity.append(fs.pi())

for gens in range(Ne):
    fs.integrate([2], 4/2/Ne)
    singletons.append(fs[1])
    doubletons.append(fs[2])
    tripletons.append(fs[3])
    diversity.append(fs.pi())

import matplotlib.pylab as plt
fig = plt.figure(1)
ax = plt.subplot(1, 1, 1)
tt = [4 * t for t in range(Ne + 1)]
ax.plot(tt, singletons / singletons[0], label="Singletons")
ax.plot(tt, doubletons / doubletons[0], label="Doubletons")
ax.plot(tt, tripletons / tripletons[0], label="Tripletons")
ax.plot(tt, diversity / diversity[0], label="Diversity (pi)")
ax.set_xlabel("Generations after expansion")
ax.legend(frameon=False)
```

```
<matplotlib.legend.Legend at 0x7f3a8c9e9390>
```



- Tajima's D and pi over time with size changes
- dynamics of allele frequency classes with size changes

11.3 Multiple populations

- Comparison to some classical result in an IM model?
- m-T confounding in heatmap of Fst
- Fst with small sizes vs large divergence
- pi over time in OOA model

CHAPTER

TWELVE

DFE INFERENCE

By Aaron Ragsdale, November 2020.

The distribution of fitness effects (DFE) for new mutations describes is a fundamental parameter in evolutionary biology - it determines the fixation probability of new functional mutations, the strength of background selection, and the genetic architecture of traits and disease.

Very roughly, most new mutations across the genome are effectively neutral or deleterious, with a small fraction being beneficial (e.g. [Keightley], [Boyko]). In coding regions, the average selection coefficient for a new mutation depends on its functional effect: we typically assume synonymous (or silent) mutations are effectively neutral (though this may be a tenuous assumption!), missense (or nonsynonymous) mutations are more deleterious on average, and loss-of-function (or nonsense) mutations are often very damaging. We can learn about the DFE in each of these categories by studying the distributions allele frequencies for variants in each class.

12.1 Data

Let's first look at the data we'll be working with. Here, I used single-population data from the Mende from Sierre Leone (MSL) from the 1000 Genomes Project [1000G]. In Fig. 12.1, I plotted the unfolded SFS for three classes of mutations in coding regions genome-wide. We can see that the missense variants are skewed to lower frequencies than synonymous variants, on average, and loss-of-function (LOF) variants are skewed to even lower frequencies.

It can be difficult to judge the skew of the SFS based on SFS counts, since the total mutational target for each mutation class differs (Table 12.1). In the bottom panel of the plot, we can see that of all LOF variants observed in the MSL population, roughly 50% of them are singletons; compare that to synonymous variants, of which less than 30% are singletons.

12.1.1 Mutation rates

The overall scaling of the SFS from mutation classes is also informative, because strongly deleterious or lethal mutations are quickly lost from the population and so are often unseen. Thus, seeing fewer mutations that expected in a given class tells us that some fraction of those mutations are highly deleterious. To make such an inference about the strongly damaging tail fo the DFE we need to know the total mutation rates for each class of mutations.

Using the mutation model from [Karczewski], I summed across all possible mutations in genes genome-wide, their mutational probability, and their functional consequences to get the total mutation rate (`u*L` - here, `L` is roughly 36 Mb of annotated coding regions) for each of the three mutation classes shown in Fig. 12.1:

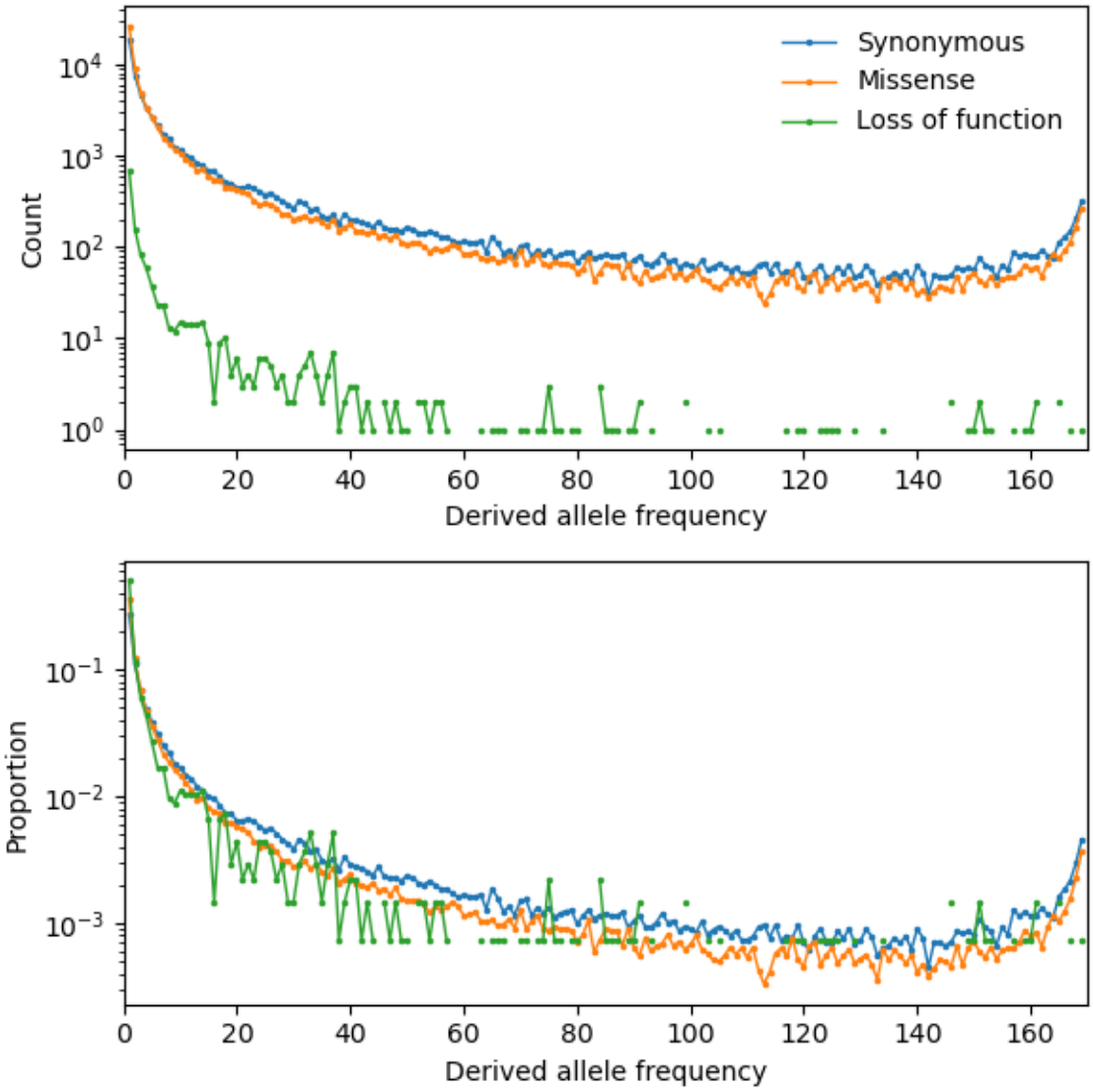


Fig. 12.1: Synonymous, missense, and loss-of-function SFS from the 1000 Genomes Project across all autosomal genes. Top: counts in each frequency bin. Bottom: proportions in each frequency bin.

Table 12.1: Total mutation rates for classes of mutations in coding regions.

Mutation class	Total mutation rate
Synonymous variants	0.1442
Missense variants	0.3426
Loss-of-function variants	0.0256

We can see here that the mutational target for nonsynonymous variants is about 2.37 times larger than for synonymous variants. Still, we see far more segregating synonymous mutations than nonsynonymous mutations:

```
import moments
import pickle
import numpy as np

# note that these frequency spectra are saved in the docs directory of the moments
# repository: https://bitbucket.org/simongravel/moments/src/master/docs/data/
data = pickle.load(open("./data/msl_data.bp", "rb"))

fs_syn = data["spectra"]["syn"]
fs_mis = data["spectra"]["mis"]
fs_lof = data["spectra"]["lof"]

u_syn = data["rates"]["syn"]
u_mis = data["rates"]["mis"]
u_lof = data["rates"]["lof"]

print("Diversity:")
print(f"synonymous:\t{fs_syn.pi():.2f}")
print(f"missense:\t{fs_mis.pi():.2f}")
print(f"loss of func:\t{fs_lof.pi():.2f}")

print()
print("Diversity scaled by total mutation rate:")
print(f"synonymous:\t{fs_syn.pi() / u_syn:.2f}")
print(f"missense:\t{fs_mis.pi() / u_mis:.2f}")
print(f"loss of func:\t{fs_lof.pi() / u_lof:.2f}")
```

```
Diversity:
synonymous:        8452.01
missense:        6991.16
loss of func:        95.16

Diversity scaled by total mutation rate:
synonymous:        58614.15
missense:        20408.81
loss of func:        3718.19
```

12.2 Controlling for demography

Demography (in this case, the population size history) affects mutation frequency trajectories and the SFS, so we need to control for non-steady-state demography in some way. Using our assumption that synonymous variants are effectively neutral, we first fit a demographic model to synonymous variants, and then with that inferred demography we fit the DFE to selected variants.

We could pick any plausible demographic model to fit. The main consideration is to choose a demographic model that can adequately fit the data, but is not so over-parameterized to be overfitting to the noise in the SFS. In Fig. 12.1, we can also see the telltale sign of ancestral misidentification by the uptick of high-frequency variants. In addition to the demographic parameters (sizes and epoch times), we wil also fit a parameter to account for the probability of mis-polarizing a variant.

Let's fit a model with three epochs: the ancestral size, an ancient expansion, and a recent exponential growth. In fitting the demography, we keep `multinom=True`, the default, as we don't have an estimate for N_e.

```
def model_func(params, ns):
    nuA, nuF, TA, TF, p_misid = params
    fs = moments.Demographics1D.snm(ns)
    fs.integrate([nuA], TA)
    nu_func = lambda t: [nuA * np.exp(np.log(nuF / nuA) * t / TF)]
    fs.integrate(nu_func, TF)
    fs = (1 - p_misid) * fs + p_misid * fs[::-1]
    return fs

p_guess = [2.0, 10.0, 0.3, 0.01, 0.02]
lower_bound = [1e-3, 1e-3, 1e-3, 1e-3, 1e-3]
upper_bound = [10, 100, 1, 1, 0.999]

opt_params = moments.Inference.optimize_log_fmin(
    p_guess, fs_syn, model_func,
    lower_bound=lower_bound, upper_bound=upper_bound)

model = model_func(opt_params, fs_syn.sample_sizes)
opt_theta = moments.Inference.optimal_sfs_scaling(model, fs_syn)
Ne = opt_theta / u_syn / 4

print("optimal demog. parameters:", opt_params[:-1])
print("anc misid:", opt_params[-1])
print("inferred Ne:", f"{Ne:.2f}")
```

```
optimal demog. parameters: [2.21531687 5.29769918 0.55450117 0.04088086]
anc misid: 0.01975812
inferred Ne: 11372.91
Log-likelihood: -689.7426549382283
```

Note that I initialized the model parameters fairly close to the optimal parameters. In practice, you would want to test a wide range of initial conditions to make sure our inference didn't get stuck at a local minimum.

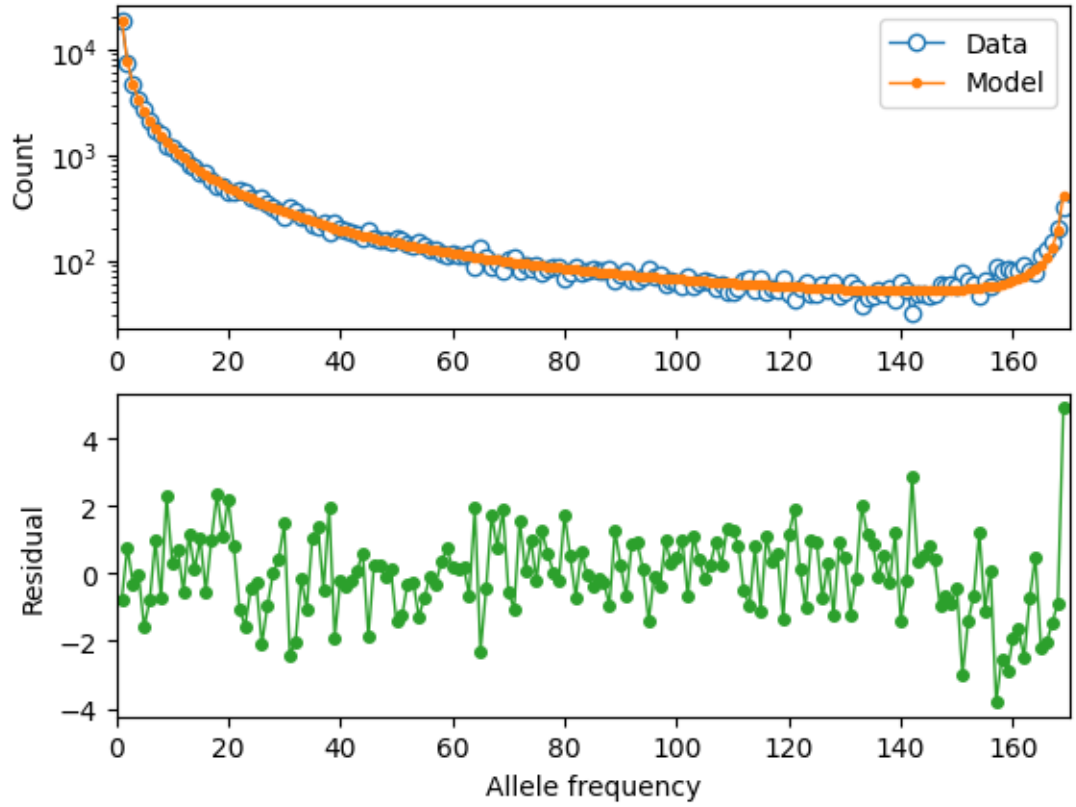
We can see how well our model fit the synonymous data:

```
moments.Plotting.plot_1d_comp_multinom(model, fs_syn, residual="linear")

# Demographic model fit to the MSL synonymous data. Top: model (red) and synonymous
```

(continues on next page)

(continued from previous page)

```
# data (blue) SFS. Bottom: residuals, plotted as ``(model - data) / sqrt(data)``.
```



That's a pretty good fit! Now that we have our inferred demographic model, let's move on to inferring the DFEs for missense and LOF variants.

12.3 Inferring the DFE

Now that we have a plausible demographic model, we can move to the selected SFS. Not every new missense mutation or every new LOF mutation will have the same fitness effect, so we aim to learn the *distribution* of selection coefficients of new mutations. Here, we are going to assume an additive model of selection - that is, heterozygotes have fitness $1 + s$ while homozygotes for the derived allele have fitness $1 + 2s$. We're also only going to focus on the deleterious DFE - we assume beneficial mutations are very rare, and we'll ignore them.

The general strategy is to pick some distribution (here, we'll choose a gamma distribution, though other distributions such a log-normal or point masses could be used), and then infer the parameters of that distribution. To do so, we compute a large number of SFS spanning the range of the distribution of possible $\gamma = 2N_e s$ values, and then combine them based on weights given by the parameterized DFE (for example, [Ragsdale], [Kim]).

Because the underlying demographic model does not change, we can cache the SFS for each value of γ. Then in optimizing the DFE parameters, we just have a weighted sum across this cache, and this makes the actual DFE inference very rapid.

12.3.1 Caching SFS

We cache the SFS for the inferred demography and a grid of selection coefficients ranging from neutral to strongly deleterious. For the SFS with very deleterious selection coefficients, the computation is only stable with large sample sizes. Thus, after each computation for a given selection coefficient, we check to make sure that the SFS does not have large negative oscillations and did not fail to converge. If the computation failed, we doube the sample size and recompute the SFS, repeating until we have a sample size large enough to stably compute the SFS. That SFS is then projected to the needed sample size and chached.

```
def selection_spectrum(gamma, h=0.5):
    rerun = True
    ns_sim = 100
    while rerun:
        ns_sim = 2 * ns_sim
        fs = moments.LinearSystem_1D.steady_state_1D(ns_sim, gamma=gamma, h=h)
        fs = moments.Spectrum(fs)
        fs.integrate([opt_params[0]], opt_params[2], gamma=gamma, h=h)
        nu_func = lambda t: [opt_params[0] * np.exp(
            np.log(opt_params[1] / opt_params[0]) * t / opt_params[3])]
        fs.integrate(nu_func, opt_params[3], gamma=gamma, h=h)
        if abs(np.max(fs)) > 10 or np.any(np.isnan(fs)):
            # large gamma-values can require large sample sizes for stability
            rerun = True
        else:
            rerun = False
    fs = fs.project(fs_syn.sample_sizes)
    return fs

spectrum_cache = {}
spectrum_cache[0] = selection_spectrum(0)

gammas = np.logspace(-4, 3, 61)
for gamma in gammas:
    spectrum_cache[gamma] = selection_spectrum(-gamma)
```

12.3.2 Optimization of the DFE

We'll fit a gamma distribution for the DFE, which has parameters alpha and beta. First, we set up the expected thetas for both missense and LOF mutations, as well as the function that weights the cached spectra based on the gamma distribution. The parameters we fit are then alpha and beta (or shape and scale) of the gamma distribution and the misidentification rate.

```
import scipy.stats
theta_mis = opt_theta * u_mis / u_syn
theta_lof = opt_theta * u_lof / u_syn

dxs = ((gammas - np.concatenate(([gammas[0]], gammas))[:-1]) / 2
    + (np.concatenate((gammas, [gammas[-1]]))[1:] - gammas) / 2)

def dfe_func(params, ns, theta=1):
    alpha, beta, p_misid = params
    fs = spectrum_cache[0] * scipy.stats.gamma.cdf(gammas[0], alpha, scale=beta)
```

(continues on next page)

(continued from previous page)

```
    weights = scipy.stats.gamma.pdf(gammas, alpha, scale=beta)
    for gamma, dx, w in zip(gammas, dxs, weights):
        fs += spectrum_cache[gamma] * dx * w
    fs = theta * fs
    return (1 - p_misid) * fs + p_misid * fs[::-1]

def model_func_missense(params, ns):
    return dfe_func(params, ns, theta=theta_mis)

def model_func_lof(params, ns):
    return dfe_func(params, ns, theta=theta_lof)
```

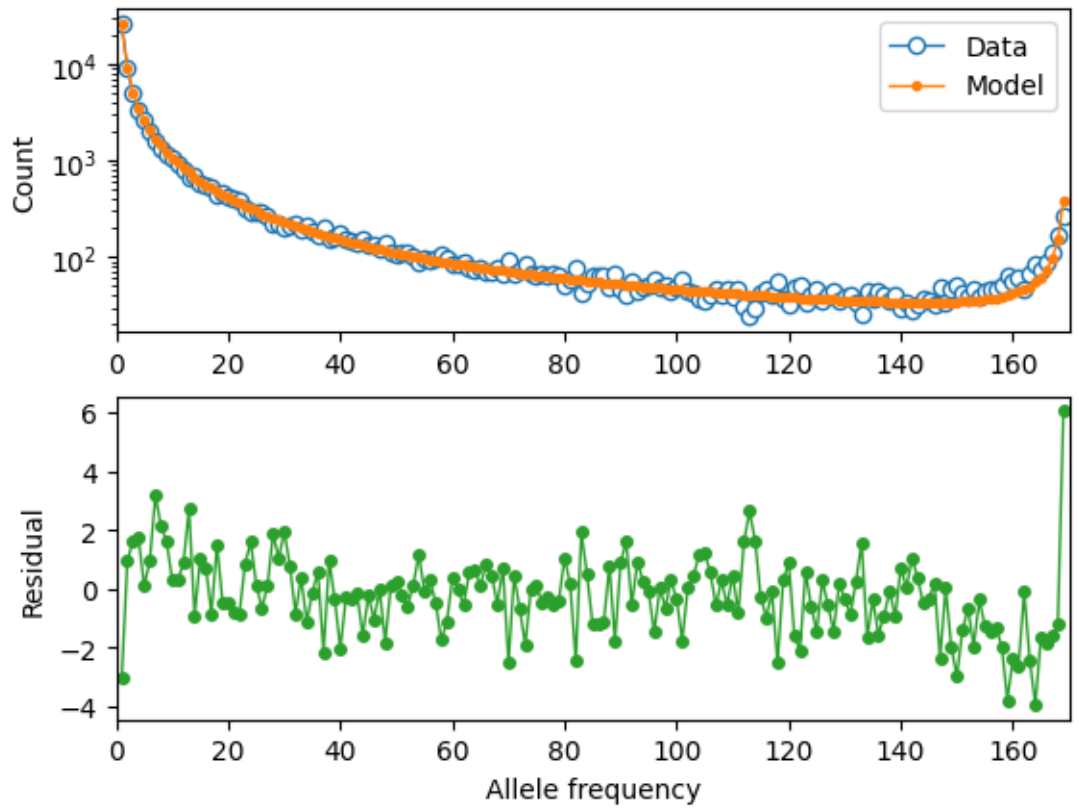
Fit missense variants:

```
p_guess = [0.2, 1000, 0.01]
lower_bound = [1e-4, 1e-1, 1e-3]
upper_bound = [1e1, 1e5, 0.999]

opt_params_mis = moments.Inference.optimize_log_fmin(
    p_guess, fs_mis, model_func_missense,
    lower_bound=lower_bound, upper_bound=upper_bound,
    multinom=False)

model_mis = model_func_missense(opt_params_mis, fs_mis.sample_sizes)
print("optimal parameters:")
print("shape:", f"{opt_params_mis[0]:.4f}")
print("scale:", f"{opt_params_mis[1]:.1f}")
print("anc misid:", f"{opt_params_mis[2]:.4f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model_mis, fs_mis))
```

```
optimal parameters:
shape: 0.1596
scale: 2332.3
anc misid: 0.0137
Log-likelihood: -695.1273435550006
```

To visualize the fit of our inferred model to the missense data:

```
moments.Plotting.plot_1d_comp_Poisson(model_mis, fs_mis, residual="linear")

# Gamma-DFE fit to the MSL missense data.
```



Next, we fit LOF variants in exactly the same way:

```
p_guess = [0.2, 1000, 0.01]
lower_bound = [1e-4, 1e-1, 1e-3]
upper_bound = [1e1, 1e5, 0.999]

opt_params_lof = moments.Inference.optimize_log_fmin(
    p_guess, fs_lof, model_func_lof,
    lower_bound=lower_bound, upper_bound=upper_bound,
    multinom=False)

model_lof = model_func_lof(opt_params_lof, fs_lof.sample_sizes)
print("optimal parameters:")
print("shape:", f"{opt_params_lof[0]:.4f}")
print("scale:", f"{opt_params_lof[1]:.1f}")
print("anc misid:", f"{opt_params_lof[2]:.4f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model_lof, fs_lof))
```

```
optimal parameters:
shape: 0.3589
scale: 7830.5
anc misid: 0.0021
Log-likelihood: -232.59479649815248
```

```
optimal parameters:
shape: 0.3589
```

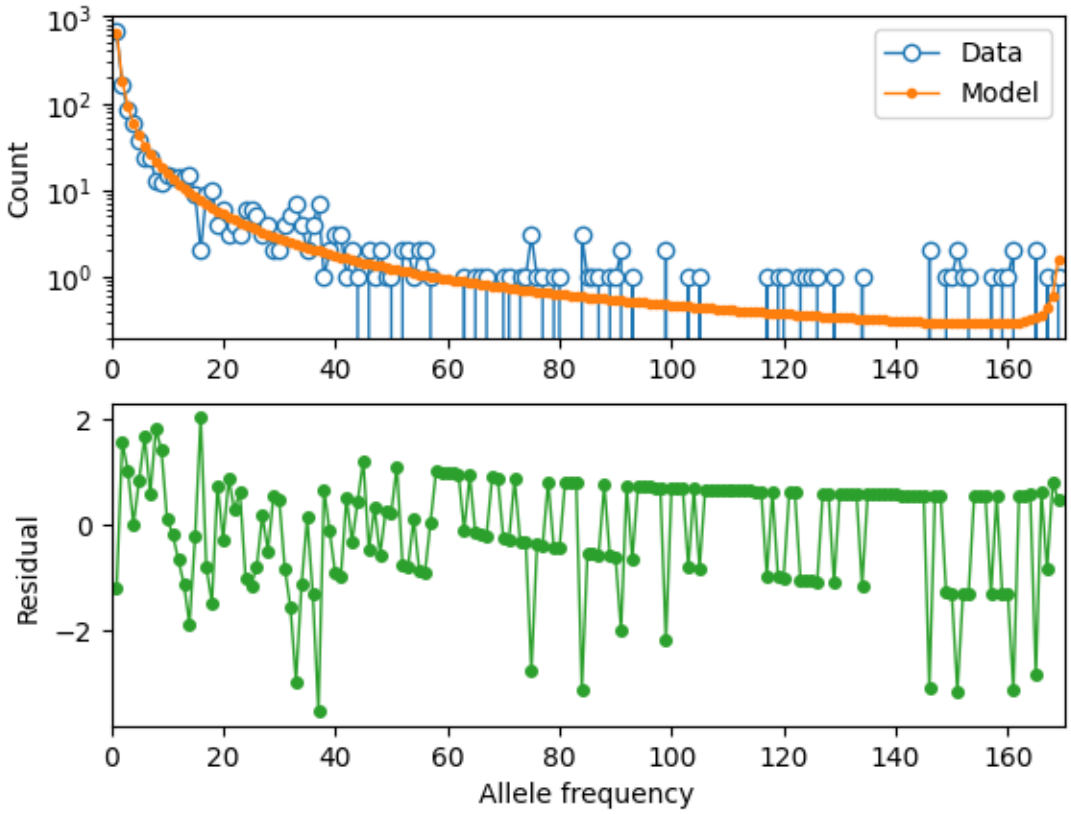
(continues on next page)

(continued from previous page)

```
scale: 7830.5
anc misid: 0.0021
Log-likelihood: -232.5947964992979
```

And again we visualize the fit of our inferred model to the LOF data:

```
moments.Plotting.plot_1d_comp_Poisson(model_lof, fs_lof, residual="linear")

# Gamma-DFE fit to the MSL loss-of-function data.
```



Using the inferred N_e from fitting the demographic model to the synonymous data and the function `scipy.stats.gamma.cdf()`, we can compute the proportions of new missense and LOF mutations across bins of selection coefficients:

Table 12.2: The DFE for missense and loss-of-function variants binned by selection coefficients, ranging from neutral or nearly neutral ($|s| < 10^{-5}$) to strongly deleterious and lethal ($|s| \geq 10^{-2}$).

Class	$\|s\| < 10^{-5}$	$10^{-5} \leq \|s\| < 10^{-4}$	$10^{-4} \leq \|s\| < 10^{-3}$	$10^{-3} \leq \|s\| < 10^{-2}$	$\|s\| \geq 10^{-2}$
Missense	0.246	0.109	0.157	0.219	0.268
LOF	0.026	0.034	0.078	0.175	0.687

Here, we clearly see that LOF variants are inferred to be very deleterious, with roughly 2/3 of all new LOF mutations having a selection coefficient larger that 10^{-2}.

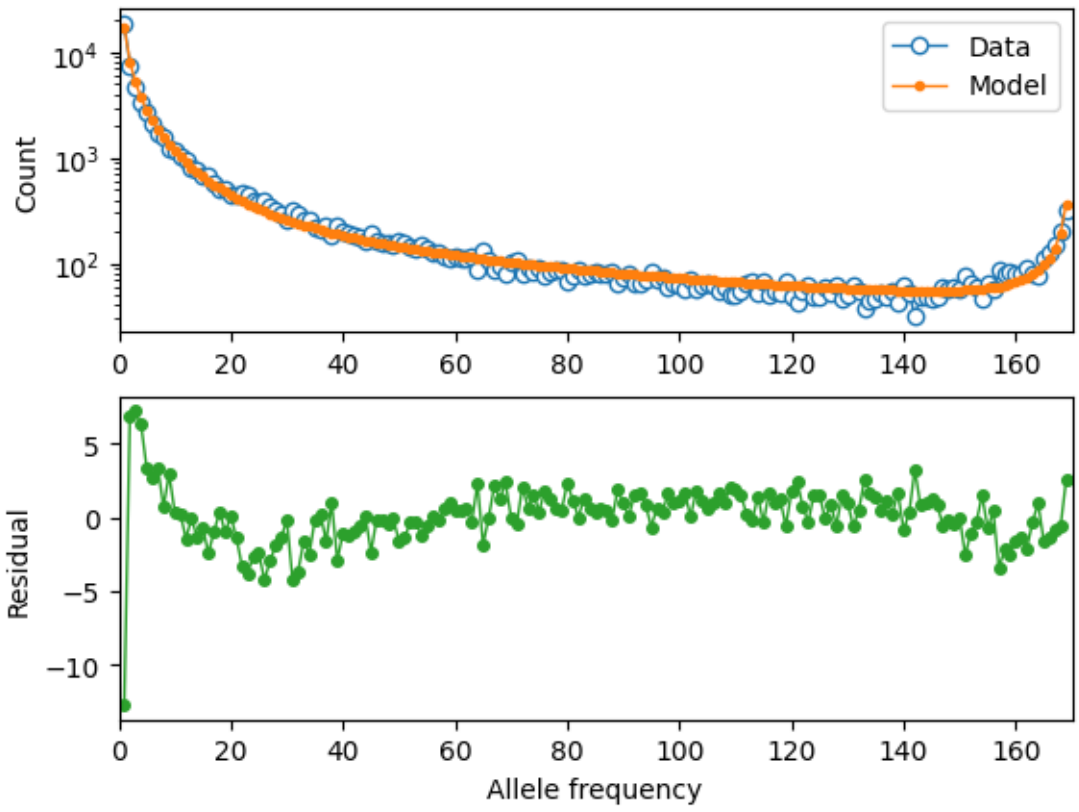
12.4 Sensitivity to the demographic model

Here, we'll fit a simpler models to the synonymous variants, and rerun the same DFE inference to check if the results are robust. We'll first fit a two-epoch model (again accounting for ancestral misidentification), and then simply use a standard neutral model without size changes.

Throughout this section, we again print log-likelihoods of the fits, which can be compared to the fits made with the more complex demographic model above.

```
def model_func(params, ns):
    nu, T, p_misid = params
    fs = moments.Demographics1D.two_epoch([nu, T], ns)
    fs = (1 - p_misid) * fs + p_misid * fs[::-1]
    return fs

p_guess = [2, .3, 0.02]
lower_bound = [1e-3, 1e-3, 1e-3]
upper_bound = [10, 1, 0.999]

opt_params = moments.Inference.optimize_log_fmin(
    p_guess, fs_syn, model_func,
    lower_bound=lower_bound, upper_bound=upper_bound)

model = model_func(opt_params, fs_syn.sample_sizes)
opt_theta = moments.Inference.optimal_sfs_scaling(model, fs_syn)
Ne = opt_theta / u_syn / 4

print("optimal demog. parameters:", opt_params[:-1])
print("anc misid:", opt_params[-1])
print("inferred Ne:", f"{Ne:.2f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model, fs_syn))
# compare log-likelihood to the more complex demographic model above

moments.Plotting.plot_1d_comp_multinom(model, fs_syn, residual="linear")
```

```
optimal demog. parameters: [2.55501781 0.31744642]
anc misid: 0.01842965
inferred Ne: 12518.37
Log-likelihood: -908.8631695023389
```



Now we cache the selection-SFS for this demography and refit the DFE to the missense variants:

```
def selection_spectrum(gamma):
    rerun = True
    ns_sim = 100
    while rerun:
        ns_sim = 2 * ns_sim
        fs = moments.LinearSystem_1D.steady_state_1D(ns_sim, gamma=gamma)
        fs = moments.Spectrum(fs)
        fs.integrate([opt_params[0]], opt_params[1], gamma=gamma)
        if abs(np.max(fs)) > 10 or np.any(np.isnan(fs)):
            # large gamma-values can require large sample sizes for stability
            rerun = True
        else:
            rerun = False
    fs = fs.project(fs_syn.sample_sizes)
    return fs

spectrum_cache = {}
spectrum_cache[0] = selection_spectrum(0)

gammas = np.logspace(-4, 3, 61)
for gamma in gammas:
    spectrum_cache[gamma] = selection_spectrum(-gamma)
```

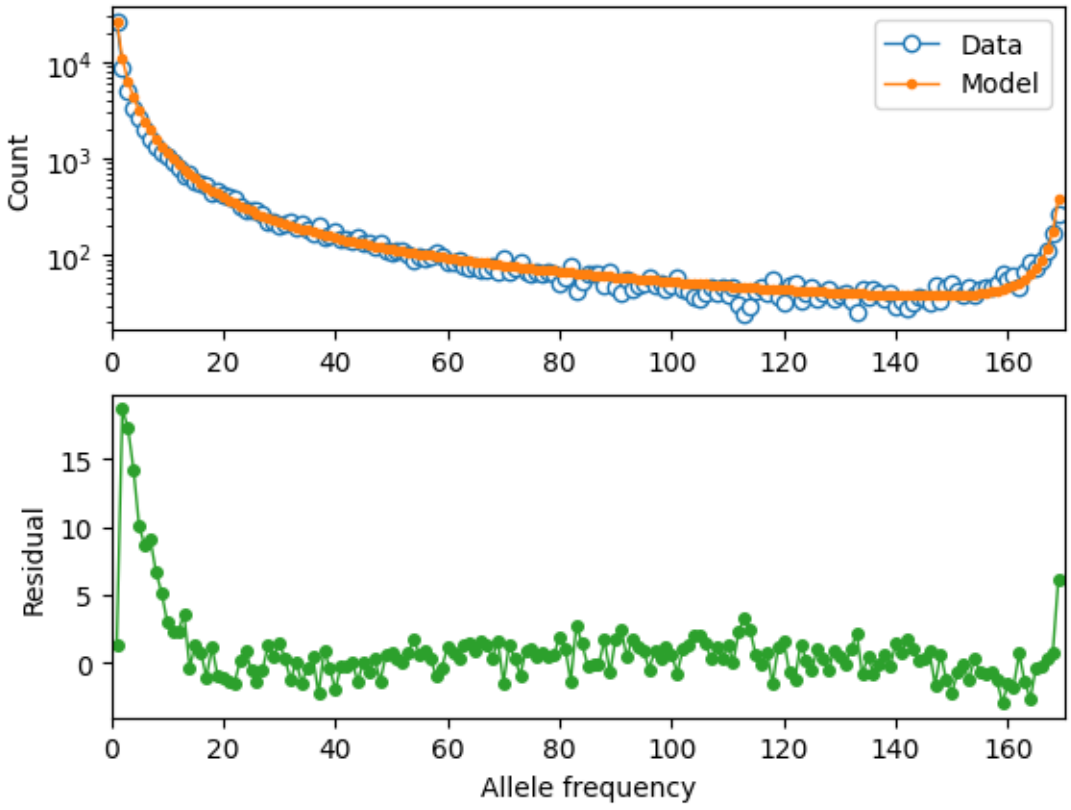
Set up the mutation rates and DFE functions:

```
theta_mis = opt_theta * u_mis / u_syn
theta_lof = opt_theta * u_lof / u_syn

dxs = ((gammas - np.concatenate(([gammas[0]], gammas))[:-1]) / 2
    + (np.concatenate((gammas, [gammas[-1]]))[1:] - gammas) / 2)

def dfe_func(params, ns, theta=1):
    alpha, beta, p_misid = params
    fs = spectrum_cache[0] * scipy.stats.gamma.cdf(gammas[0], alpha, scale=beta)
    weights = scipy.stats.gamma.pdf(gammas, alpha, scale=beta)
    for gamma, dx, w in zip(gammas, dxs, weights):
        fs += spectrum_cache[gamma] * dx * w
    fs = theta * fs
    return (1 - p_misid) * fs + p_misid * fs[::-1]

def model_func_missense(params, ns):
    return dfe_func(params, ns, theta=theta_mis)

def model_func_lof(params, ns):
    return dfe_func(params, ns, theta=theta_lof)
```

Fit the missense data:

```
p_guess = [0.2, 1000, 0.01]
lower_bound = [1e-4, 1e-1, 1e-3]
upper_bound = [1e1, 1e5, 0.999]

opt_params_mis = moments.Inference.optimize_log_fmin(
    p_guess, fs_mis, model_func_missense,
    lower_bound=lower_bound, upper_bound=upper_bound,
    multinom=False)

model_mis = model_func_missense(opt_params_mis, fs_mis.sample_sizes)
print("optimal parameters (missense):")
print("shape:", f"{opt_params_mis[0]:.4f}")
print("scale:", f"{opt_params_mis[1]:.1f}")
print("anc misid:", f"{opt_params_mis[2]:.4f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model_mis, fs_mis))

moments.Plotting.plot_1d_comp_Poisson(model_mis, fs_mis, residual="linear")
```

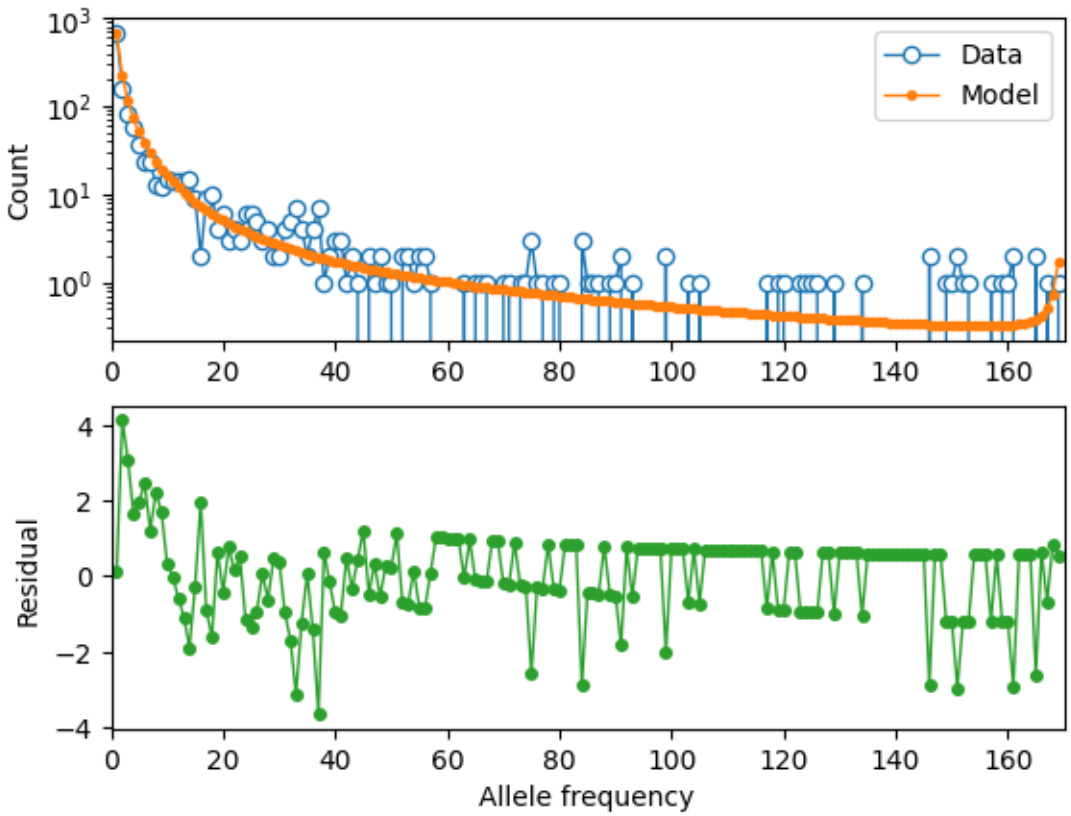
```
optimal parameters (missense):
shape: 0.1830
scale: 733.6
anc misid: 0.0134
Log-likelihood: -999.0833946058101
```



Fit the LOF data:

```
p_guess = [0.2, 1000, 0.01]
lower_bound = [1e-4, 1e-1, 1e-3]
upper_bound = [1e1, 1e5, 0.999]

opt_params_lof = moments.Inference.optimize_log_fmin(
    p_guess, fs_lof, model_func_lof,
    lower_bound=lower_bound, upper_bound=upper_bound,
    multinom=False)

model_lof = model_func_lof(opt_params_lof, fs_lof.sample_sizes)
print("optimal parameters:")
print("shape:", f"{opt_params_lof[0]:.4f}")
print("scale:", f"{opt_params_lof[1]:.1f}")
print("anc misid:", f"{opt_params_lof[2]:.4f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model_lof, fs_lof))

moments.Plotting.plot_1d_comp_Poisson(model_lof, fs_lof, residual="linear")
```

```
optimal parameters:
shape: 0.3937
scale: 3802.9
anc misid: 0.0021
Log-likelihood: -246.78811141192315
```



We can compare our results using this simpler two-epoch demographic model to our previous findings:

```
print("Missense DFE:")
shape = opt_params_mis[0]
scale = opt_params_mis[1]
ss = [0, 1e-5, 1e-4, 1e-3, 1e-2]
for s0, s1 in zip(ss[:-1], ss[1:]):
    cdf0 = scipy.stats.gamma.cdf(2 * Ne * s0, shape, scale=scale)
    cdf1 = scipy.stats.gamma.cdf(2 * Ne * s1, shape, scale=scale)
    print(f"{s0} <= s < {s1}:", f"{cdf1 - cdf0:.3f}")
    if s1 == ss[-1]:
        print(f"s >= {s1}:", f"{1 - cdf1:.3f}")

print()
print("LOF DFE:")
shape = opt_params_lof[0]
scale = opt_params_lof[1]
ss = [0, 1e-5, 1e-4, 1e-3, 1e-2]
for s0, s1 in zip(ss[:-1], ss[1:]):
    cdf0 = scipy.stats.gamma.cdf(2 * Ne * s0, shape, scale=scale)
    cdf1 = scipy.stats.gamma.cdf(2 * Ne * s1, shape, scale=scale)
    print(f"{s0} <= s < {s1}:", f"{cdf1 - cdf0:.3f}")
    if s1 == ss[-1]:
        print(f"s >= {s1}:", f"{1 - cdf1:1.3f}")
```

```
Missense DFE:
```

(continues on next page)

(continued from previous page)

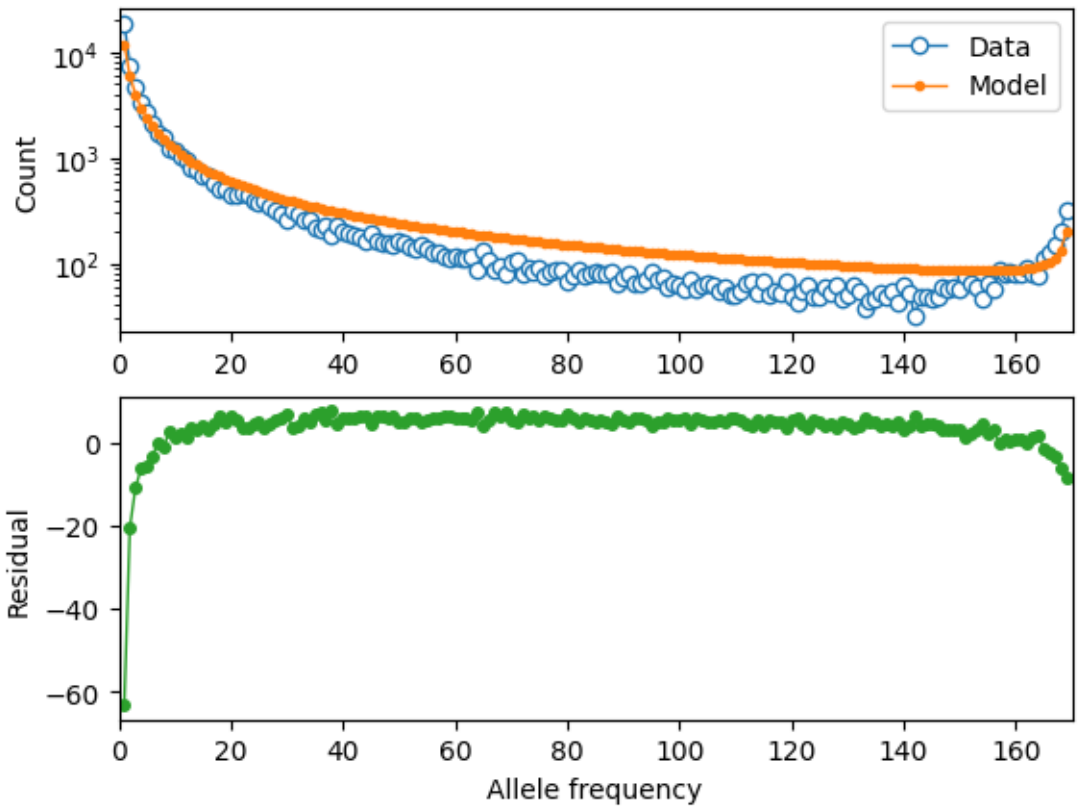
```
0 <= s < 1e-05: 0.251
1e-05 <= s < 0.0001: 0.132
0.0001 <= s < 0.001: 0.198
0.001 <= s < 0.01: 0.266
s >= 0.01: 0.153

LOF DFE:
0 <= s < 1e-05: 0.025
1e-05 <= s < 0.0001: 0.038
0.0001 <= s < 0.001: 0.093
0.001 <= s < 0.01: 0.223
s >= 0.01: 0.621
```

Comparing to the table above, these look pretty similar - that's a good sign that our inferences are fairly robust to slightly poorer fits of the demographic model.

But what if our demographic model is way off, such as assuming constant population size?

```
# here, we'll only fit the ancestral-state misidentification rate
def model_func(params, ns):
    p_misid = params
    fs = moments.Demographics1D.snm(ns)
    fs = (1 - p_misid) * fs + p_misid * fs[::-1]
    return fs

p_guess = [0.02]
lower_bound = [1e-3]
upper_bound = [0.999]

opt_params = moments.Inference.optimize_log_fmin(
    p_guess, fs_syn, model_func,
    lower_bound=lower_bound, upper_bound=upper_bound)

model = model_func(opt_params, fs_syn.sample_sizes)
opt_theta = moments.Inference.optimal_sfs_scaling(model, fs_syn)
Ne = opt_theta / u_syn / 4

print("optimal Ne scaling:", f"{Ne:.2f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model, fs_syn))

moments.Plotting.plot_1d_comp_multinom(model, fs_syn, residual="linear")
```

```
optimal Ne scaling: 20930.55
Log-likelihood: -4856.925204009832
```



Set up the spectrum cache for this constant-size demographic model:

```
def selection_spectrum(gamma):
    fs = moments.LinearSystem_1D.steady_state_1D(fs_syn.sample_sizes[0], gamma=gamma)
    fs = moments.Spectrum(fs)
    return fs

spectrum_cache = {}
spectrum_cache[0] = selection_spectrum(0)

gammas = np.logspace(-4, 3, 61)
for gamma in gammas:
    spectrum_cache[gamma] = selection_spectrum(-gamma)
```

Set up the mutation rates and DFE functions:

```
theta_mis = opt_theta * u_mis / u_syn
theta_lof = opt_theta * u_lof / u_syn

dxs = ((gammas - np.concatenate(([gammas[0]], gammas))[:-1]) / 2
    + (np.concatenate((gammas, [gammas[-1]]))[1:] - gammas) / 2)

def dfe_func(params, ns, theta=1):
    alpha, beta, p_misid = params
    fs = spectrum_cache[0] * scipy.stats.gamma.cdf(gammas[0], alpha, scale=beta)
    weights = scipy.stats.gamma.pdf(gammas, alpha, scale=beta)
    for gamma, dx, w in zip(gammas, dxs, weights):
```

(continues on next page)

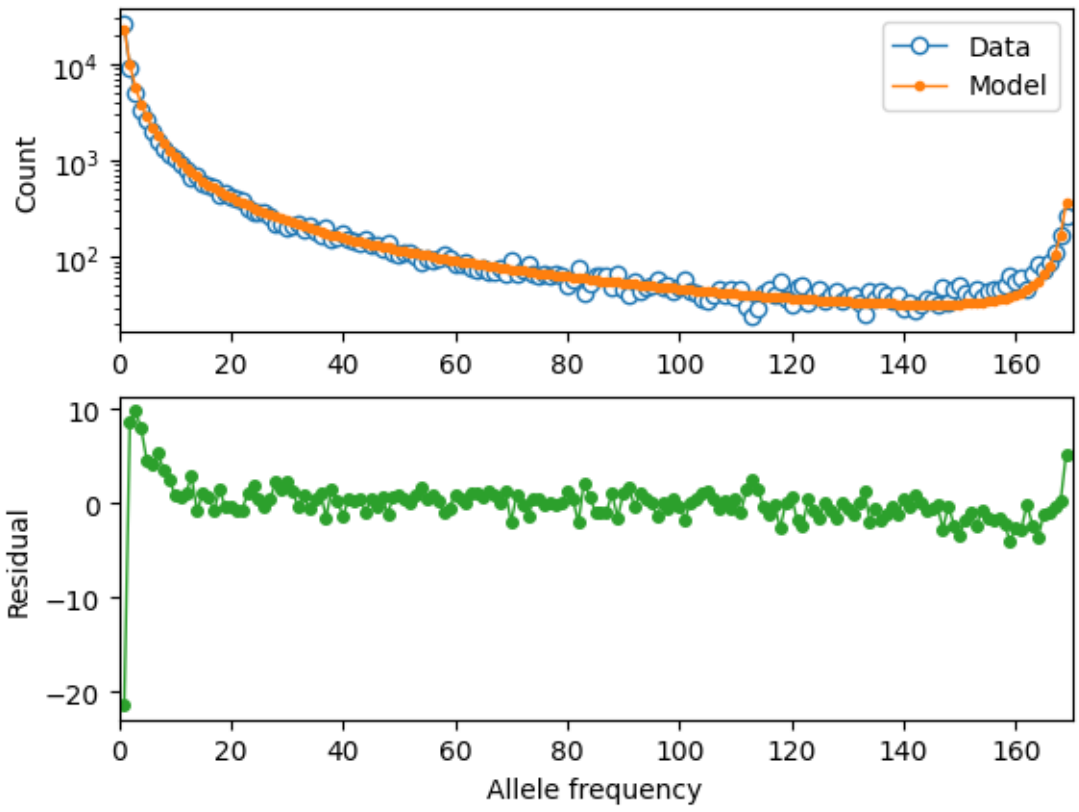
(continued from previous page)

```
        fs += spectrum_cache[gamma] * dx * w
    fs = theta * fs
    return (1 - p_misid) * fs + p_misid * fs[::-1]

def model_func_missense(params, ns):
    return dfe_func(params, ns, theta=theta_mis)

def model_func_lof(params, ns):
    return dfe_func(params, ns, theta=theta_lof)
```

Fit the missense data:

```
p_guess = [0.2, 1000, 0.01]
lower_bound = [1e-4, 1e-1, 1e-3]
upper_bound = [1e1, 1e5, 0.999]

opt_params_mis = moments.Inference.optimize_log_fmin(
    p_guess, fs_mis, model_func_missense,
    lower_bound=lower_bound, upper_bound=upper_bound,
    multinom=False)

model_mis = model_func_missense(opt_params_mis, fs_mis.sample_sizes)
print("optimal parameters (missense):")
print("shape:", f"{opt_params_mis[0]:.4f}")
print("scale:", f"{opt_params_mis[1]:.1f}")
print("anc misid:", f"{opt_params_mis[2]:.4f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model_mis, fs_mis))

moments.Plotting.plot_1d_comp_multinom(model_mis, fs_mis, residual="linear")
```

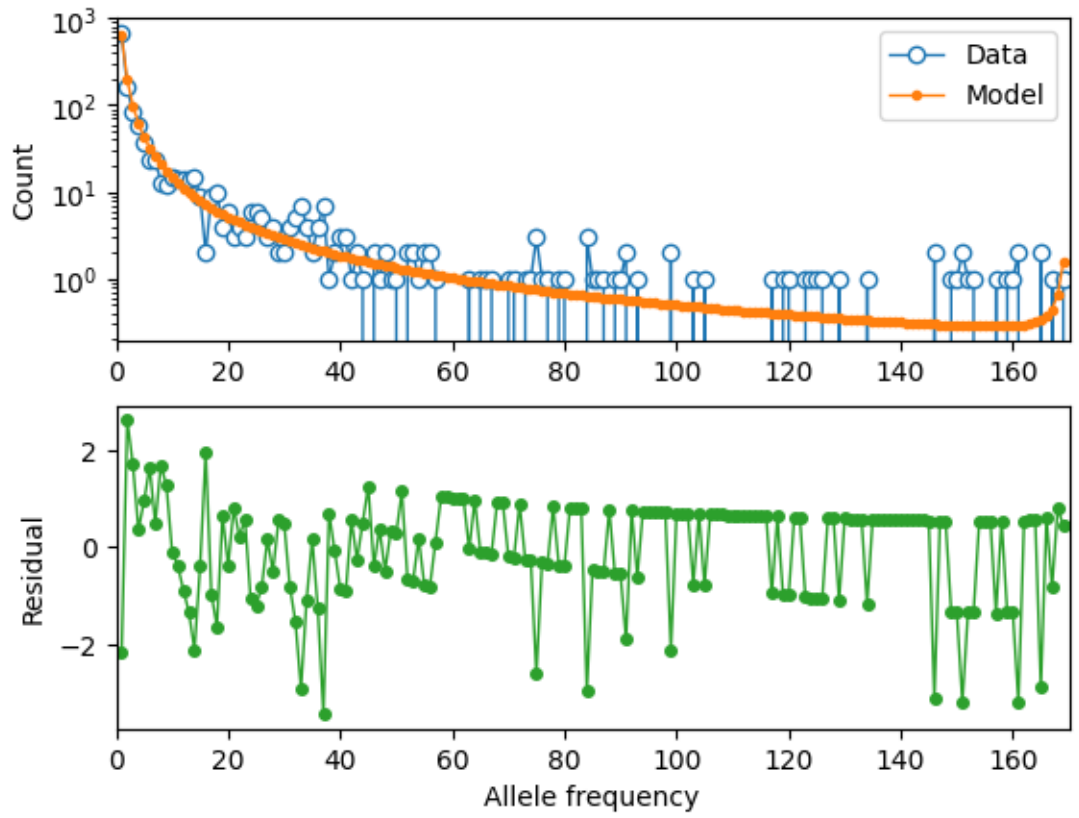
```
optimal parameters (missense):
shape: 0.4448
scale: 82.4
anc misid: 0.0147
Log-likelihood: -1065.0798831623356
```



Fit the LOF data:

```
p_guess = [0.2, 1000, 0.01]
lower_bound = [1e-4, 1e-1, 1e-3]
upper_bound = [1e1, 1e5, 0.999]

opt_params_lof = moments.Inference.optimize_log_fmin(
    p_guess, fs_lof, model_func_lof,
    lower_bound=lower_bound, upper_bound=upper_bound,
    multinom=False)

model_lof = model_func_lof(opt_params_lof, fs_lof.sample_sizes)
print("optimal parameters:")
print("shape:", f"{opt_params_lof[0]:.4f}")
print("scale:", f"{opt_params_lof[1]:.1f}")
print("anc misid:", f"{opt_params_lof[2]:.4f}")
print("Log-likelihood:", moments.Inference.ll_multinom(model_lof, fs_lof))

moments.Plotting.plot_1d_comp_multinom(model_lof, fs_lof, residual="linear")
```

```
optimal parameters:
shape: 0.5298
scale: 1601.8
anc misid: 0.0021
Log-likelihood: -237.79096099886482
```



And now comparing our results using the standard neutral model as the underlying demography:

```
print("Missense DFE:")
shape = opt_params_mis[0]
scale = opt_params_mis[1]
ss = [0, 1e-5, 1e-4, 1e-3, 1e-2]
for s0, s1 in zip(ss[:-1], ss[1:]):
    cdf0 = scipy.stats.gamma.cdf(2 * Ne * s0, shape, scale=scale)
    cdf1 = scipy.stats.gamma.cdf(2 * Ne * s1, shape, scale=scale)
    print(f"{s0} <= s < {s1}:", cdf1 - cdf0)
    if s1 == ss[-1]:
        print(f"s >= {s1}:", 1 - cdf1)

print()
print("LOF DFE:")
shape = opt_params_lof[0]
scale = opt_params_lof[1]
ss = [0, 1e-5, 1e-4, 1e-3, 1e-2]
for s0, s1 in zip(ss[:-1], ss[1:]):
    cdf0 = scipy.stats.gamma.cdf(2 * Ne * s0, shape, scale=scale)
    cdf1 = scipy.stats.gamma.cdf(2 * Ne * s1, shape, scale=scale)
    print(f"{s0} <= s < {s1}:", cdf1 - cdf0)
    if s1 == ss[-1]:
        print(f"s >= {s1}:", 1 - cdf1)
```

```
Missense DFE:
```

(continues on next page)

(continued from previous page)

```
0 <= s < 1e-05: 0.10753022385487969
1e-05 <= s < 0.0001: 0.18776852548601836
0.0001 <= s < 0.001: 0.42679335393077944
0.001 <= s < 0.01: 0.27674764907250127
s >= 0.01: 0.0011602476558212338

LOF DFE:
0 <= s < 1e-05: 0.01423964365346152
1e-05 <= s < 0.0001: 0.033952075858474665
0.0001 <= s < 0.001: 0.11371870531196496
0.001 <= s < 0.01: 0.34510157846413436
s >= 0.01: 0.49298799671196447
```

These distributions look quite different - in particular, both the missense and LOF variants are inferred to be much more deleterious. This is because we did not account for population size expansions in it history, which leads to an excess of rare variants for each class of mutations, and the model over-compensates for the excess of rare variants by fitting a DFE that is more skewed toward larger selection coefficients.

12.5 References

CHAPTER

THIRTEEN

LINKAGE DISEQUILIBRIUM AND RECOMBINATION

Todo: This module has not been completed.

13.1 Sections

- Recombination and low-order LD statistics
- Ohta and Kimura
- Single population LD decay curves
- Multiple populations
- Selfing

CHAPTER

FOURTEEN

SELECTION AT TWO LOCI

By Aaron Ragsdale, January 2021.

Note: This module has not been completed - I've placed to-dos where content is incoming. If you find an error here, or find some aspects confusing, please don't hesitate to get in touch or open an issue. Thanks!

Most users of `moments` will be most interested in computing the single-site SFS and comparing it to data. However, `moments` can do much more, such as computing expectations for LD under complex demography, or triallelic or two-locus frequency spectra. Here, we'll explore what we can do with the two-locus methods available in `moments.TwoLocus`.

```
import moments.TwoLocus
import numpy as np
import matplotlib.pylab as plt
import pickle, gzip
```

14.1 The two-locus allele frequency spectrum

Similar to the single-site SFS, the two-locus frequency spectrum stores the number (or density) of pairs of loci with given two-locus haplotype counts. Suppose the left locus permits alleles *A*/*a* and the right locus permits *B*/*b*, so that there are four possible haplotypes: (*AB*, *Ab*, *aB*, and *ab*). In a sample size of *n* haploid samples, we observe some number of each haplotype, $n_{AB} + n_{Ab} + n_{aB} + n_{ab} = n$. The two-locus frequency spectrum stores the observed number of pairs of loci with each possible sampling configuration, so that $\Psi_n(i, j, k)$ is the number (or density) of pairs of loci with *i* type *AB*, *j* type *Ab*, and *k* type *aB*.

`moments.TwoLocus` lets us compute the expectation of Ψ_n for single-population demographic scenarios, allowing for population size changes over time, as well as arbitrary recombination distance separating the two loci and selection at one or both loci. While `moments.TwoLocus` has a reversible mutation model implemented, here we'll focus on the infinite sites model (ISM), under the assumption that $N_e\mu \ll 1$ at both loci.

Below, we'll walk through how to compute the sampling distribution for two-locus haplotypes for a given sample size, describe its relationship to common measures of linkage disequilibrium (LD), and explore how recombination, demography, and selection interacts to alter expected patterns of LD. In particular, we'll focus on a few different models of selection, dominance, and epistasic interactions between loci, and ask under what conditions those patterns are expected to differ or to be confounded.

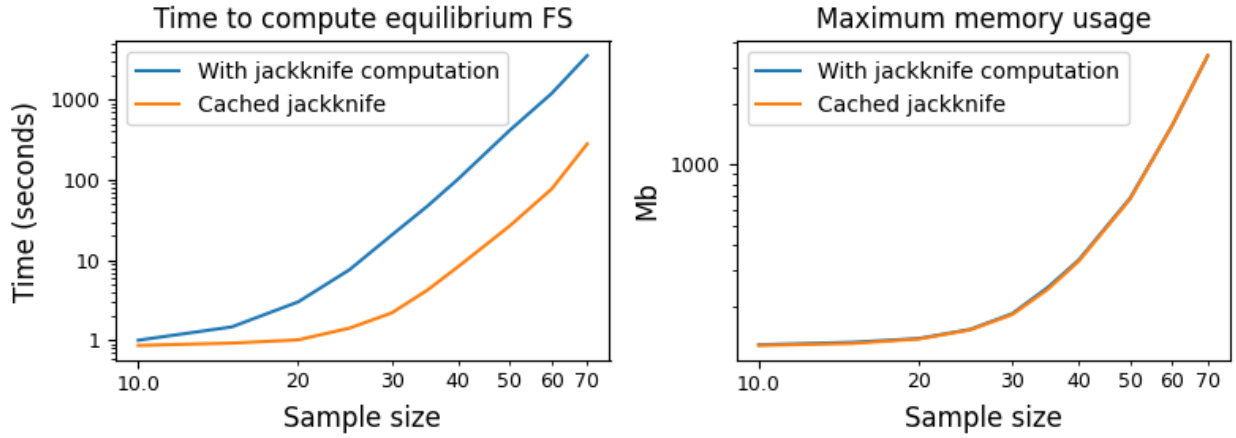
14.1.1 Citing this work

Demographic inference using a diffusion approximation-based solution for Ψ_n was introduced in [Ragsdale_Gutenkunst]. The moments-based method, which is implemented here, was described in [Ragsdale_Gravel].

14.2 Two-locus haplotype distribution under neutrality

14.2.1 A quick comment on computational efficiency

The frequency spectrum Ψ_n is displayed as a 3-dimensional array in *moments*, and the size grows quite quickly in the sample size n. (The number of frequency bins is $\frac{1}{6}(n+1)(n+2)(n+3)$, so it grows as n^3.) Thus, solving for Ψ gets quite expensive for large sample sizes.



Here, we see the time needed to compute the equilibrium frequency spectrum for a given sample size. Recombination requires computing a jackknife operator for approximate moment closure, which gets expensive for large sample sizes. However, we can cache and reuse this jackknife matrix (the default behavior), so that much of the computational time is saved from having to recompute that large matrix. However, we see that simply computing the steady-state solution still gets quite expensive as the sample sizes increase.

Below, we'll see that for non-zero recombination (as well as selection) our accuracy improves as we increase the sample size. For this reason, we've pre-computed and cached results throughout this page, and the code blocks give examples of how those results were created.

14.2.2 Two neutral loci

The `moments.TwoLocus` solution for the neutral frequency spectrum without recombination ($\rho = 4N_e r = 0$) is exact, while $\rho > 0$ and selection require a moment-closure approximation. This approximation grows more accurate for larger n.

To get familiar with some common two-locus statistics (either summaries of Ψ_n and Ψ itself), we can compare to some classical results, such as the expectation for $\sigma_d^2 = \frac{\mathbb{E}[D^2]}{\mathbb{E}[p(1-p)q(1-q)]}$, where D is the standard covariance measure of LD, and p and q are allele frequencies at the left and right loci, respectively [Ohta]:

```
rho = 0
n = 10
Psi = moments.TwoLocus.Demographics.equilibrium(n, rho=rho)
sigma_d2 = Psi.D2() / Psi.pi2()
```

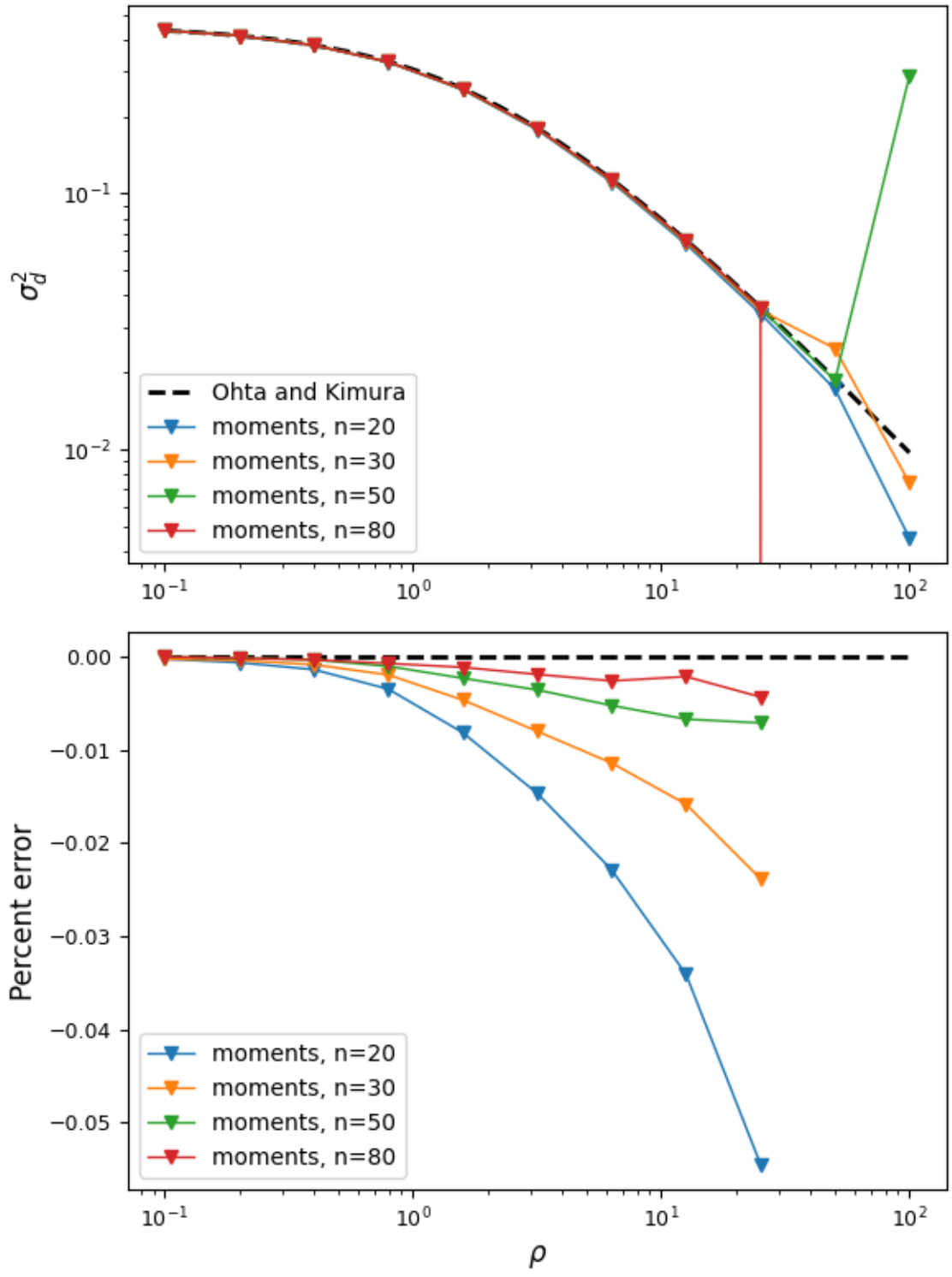
(continues on next page)

(continued from previous page)

```
print(r"moments.TwoLocus $\sigma_d^2$, $r=0$:", sigma_d2)
print(r"Ohta and Kimura expectation, $r=0$:", 5 / 11)
```

```
moments.TwoLocus $\sigma_d^2$, $r=0$: 0.4545454545454553
Ohta and Kimura expectation, $r=0$: 0.45454545454545453
```

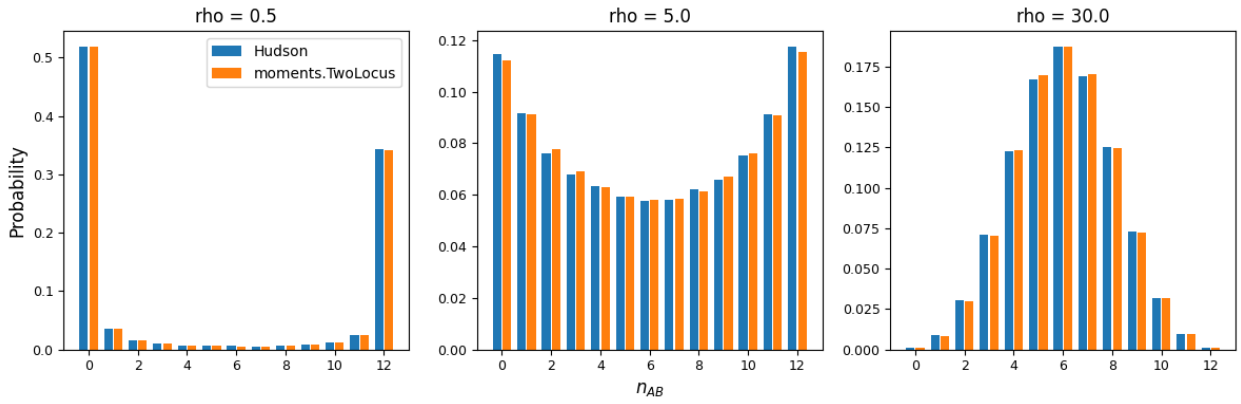
And we can plot the LD-decay curve for σ_d^2 for a range of recombination rates and a few different sample sizes, and compare to [Ohta]'s expectation, which is $\sigma_d^2 = \frac{5+\frac{1}{2}\rho}{11+\frac{13}{2}\rho+\frac{1}{2}\rho^2}$:



The moments approximation breaks down for recombination rates around $\rho \approx 50$ but is very accurate for lower recombination rates, and this accuracy increases with the sample size. To be safe, we can assume that numerical error starts to creep in around $rho \approx 25$, which for human parameters, is very roughly 50 or 100kb. So we're limited to looking at LD in relatively shorter regions. For higher recombination rates, we can turn to `moments.LD`, which lets us model multiple populations, but is restricted to neutral loci and low-order statistics.

The statistics $\mathbb{E}[D^2]$ and $\mathbb{E}[p(1-p)q(1-q)]$ are low-order summaries of the full sampling distribution, similar to how heterozygosity or Tajima's D are low-order summaries of the single-site SFS. We can visualize some features of the full two-locus haplotype frequency distribution instead, following Figure 1 in Hudson's classical paper on the two-locus sampling distribution [Hudson]. Here, we'll look at a slice in the 3-dimensional distribution: if we observe n_A samples carrying A at the left locus, and n_B carrying B at the right locus, what is the probability that we observe n_{AB} haplotypes with A and B coupled in the same sample? This marginal distribution will depend on ρ:

```
rhos = [0.5, 5.0, 30.0]
n = 30
nA = 15
nB = 12

# first we'll get the slice for the given frequencies from the "hnrho" file
# from RR Hudson: http://home.uchicago.edu/~rhudson1/source/twolocus.html
hudson = {}
import gzip
with gzip.open("./data/h30rho.gz", "rb") as fin:
    at_frequencies = False
    for line in fin:
        l = line.decode()
        if "freq" in l:
            if int(l.split()[1]) == nA and int(l.split()[2]) == nB:
                at_frequencies = True
            else:
                at_frequencies = False
        if at_frequencies:
            rho = float(l.split()[1])
            if rho in rhos:
                    hudson[rho] = np.array([float(v) for v in l.split()[2:]])

fig = plt.figure(figsize=(12, 4))
for ii, rho in enumerate(rhos):
    # results are cached, having used the following line to create the spectra
    # F = moments.TwoLocus.Demographics.equilibrium(n, rho=rho)
    F = pickle.load(gzip.open(f"./data/two-locus/eq.n_{n}.rho_{rho}.fs.gz", "rb"))
    counts, pAB = moments.TwoLocus.Util.pAB(F, nA, nB)
    pAB /= pAB.sum()
    ax = plt.subplot(1, 3, ii + 1)
    ax.bar(counts - 0.2, hudson[rho] / hudson[rho].sum(), width=0.35, label="Hudson")
    ax.bar(counts + 0.2, pAB, width=0.35, label="moments.TwoLocus")
    ax.set_title(f"rho = {rho}")
    if ii == 0:
        ax.set_ylabel("Probability")
        ax.legend()
    if ii == 1:
        ax.set_xlabel(r"$n_{AB}$")
fig.tight_layout()
```



For low recombination rates, the marginal distribution of *AB* haplotypes is skewed toward the maximum or minimum number of copies, resulting in higher LD, while for larger recombination rates, the distribution of n_{AB} is concentrated around frequencies that result in low levels of LD. We can also see that `moments.TwoLocus` agrees well with Hudson's results under neutrality and steady state demography.

Note: Below, we'll be revisiting these same statistics and seeing how various models of selection at the two loci, as well as non-steady state demography, distort the expected distributions.

14.3 How does selection interact across multiple loci?

There has been a recent resurgence of interest in learning about the interaction of selection at two or more loci (e.g., for studies within the past few years, see [Sohail], [Garcia], [Sandler], [Good]). This has largely been driven by the relatively recent availability of large-scale sequencing datasets that allow us to observe patterns of allele frequencies and LD for negatively selected loci that may be segregating at very low frequencies in a population. Some of these studies are theory-driven (e.g., [Good]), while others rely on forward Wright-Fisher simulators (such as `SLiM` or `fwdpy11`) to compare observed patterns between data and simulation.

These approaches have their limitations: analytical results are largely constrained to simple selection scenarios and steady-state demography, while simulation studies are computationally expensive and thus often end up limited to still a handful of selection and demographic scenarios. Numerical approaches to compute expectations of statistics of interest could therefore provide a far more efficient way to compute explore parameter regimes and compare model expectations to data in inference frameworks.

Here, we'll explore a few selection models, including both dominance and epistatic effects, that theory predicts should result in different patterns of LD between two selected loci. We first describe the selection models, and then we compare their expected patterns of LD.

14.3.1 Selection models at two loci

At a single locus, the effects of selection and dominance are captured by the selection coefficient s and the dominance coefficient h, so that fitnesses of the diploid genotypes are given by

Table 14.1: Single-locus fitnesses.

Genotype	Relative fitness
aa	1
Aa	$1 + 2hs$
AA	$1 + 2s$

If $h = 1/2$, i.e. selection is *additive*, this model reduces to a haploid selection model where genotype *A* has relative fitness $1 + s$ compared to a.

Additive selection, no epistasis

Additive selection models for two loci, like in the single-locus case, reduce to haploid-based models, where we only need to know the relative fitnesses of the two-locus haplotypes *AB*, *Ab*, *aB*, and *ab*. When we say "no epistasis," we typically mean that the relative fitness of an individual carrying both derived alleles (*AB*) is additive across loci, so that if s_A is the selection coefficient at the left (*A*/*a*) locus, and s_B is the selection coefficient at the right (*B*/*b*) locus, then $s_{AB} = s_A + s_B$.

Table 14.2: No epistasis or dominance emits a haploid selection model.

Genotype	Relative fitness
ab	1
Ab	$1 + s_A$
aB	$1 + s_B$
AB	$1 + s_{AB} = 1 + s_A + s_B$

Additive selection with epistasis

Epistasis is typically modeled as a factor ϵ that either increases or decreases the selection coefficient for the *AB* haplotype, so that $s_{AB} = s_A + s_B + \epsilon$. If $|s_{AB}| > |s_A| + |s_A|$, i.e. the fitness effect of the *AB* haplotype is greater than the sum of the effect of the *Ab* and *aB* haplotypes, the effect is called *synergistic* epistasis, and if $|s_{AB}| < |s_A| + |s_A|$, it is refered to as *antagonistic* epistasis.

Table 14.3: A haploid selection model with epistasis.

Genotype	Relative fitness
ab	1
Ab	$1 + s_A$
aB	$1 + s_B$
AB	$1 + s_{AB} = 1 + s_A + s_B + \epsilon$

Simple dominance, no epistasis

Epistasis is the non-additive interaction of selective effects across loci. The non-additive effect of selection within a locus is called dominance, when $s_{AA} \neq 2s_{Aa}$. Without epistasis, so that $s_{AB} = s_A + s_B$, and allowing for different selection and dominance coefficients at the two loci, the fitness effects for two-locus diploid genotypes takes a simple form analogous to the single-locus case with dominance. Here, we define the relative fitnesses of two-locus diploid genotypes, which relies on the selection and dominance coefficients at the left and right loci:

Table 14.4: Accounting for dominance requires modeling selection for diploid genotypes, instead of the model reducing to selection on haploid genotypes.

Genotype	Relative fitness
aabb	1
Aabb	$1 + 2h_A s_A$
AAbb	$1 + 2s_A$
aaBb	$1 + 2h_B s_B$
AaBb	$1 + 2h_A s_A + 2h_B s_B$
AABb	$1 + 2s_A + 2h_B s_B$
aaBB	$1 + 2s_B$
AaBB	$1 + 2h_A s_A + 2s_B$
AABB	$1 + 2s_A + 2s_B$

Both dominance and epistasis

As additional non-additive interactions are introduced, it gets more difficult to succinctly define general selection models with few parameters. A general selection model that is flexible could simply define a selection coefficient for each two-locus diploid genotype, in relation to the double wild-type homozygote (*aabb*). That is, define s_{Aabb} as the selection coefficient for the *Aabb* genotype, s_{AaBb} the selection coefficient for the *AaBb* genotype, and so on.

Gene-based dominance

In the above model, fitness is determined by combined hetero-/homozygosity at the two loci, but it does not make a distinction between the different ways that double heterozygotes (*AaBb*) could arise. Instead, we could imagine a model where diploid individual fitnesses depend on the underlying haplotypes, i.e. whether selected mutations at the two loci are coupled on the same background or are on different haplotypes.

For example, consider loss-of-function mutations in coding regions. Such mutations tend to be severely damaging. We could think of the situation where diploid individual fitness is strongly reduced when both copies carry a loss-of-function mutation, but much less reduced if the individual has at least one copy without a mutation. In this scenario, the haplotype combination *Ab / aB* will confer more reduced fitness compared to the combination *AB / ab*, even though both are double heterozygote genotypes.

Perhaps the simplest model for gene-based dominance assumes that derived mutations at the two loci (*A* and *B*) carry the same fitness cost, and fitness depends on the number of haplotype copies within a diploid individual that have at least one such mutation. This model requires just two parameters, a single selection coefficient *s* and a single dominance coefficient *h*:

Table 14.5: A simple gene-based dominance model.

Genotype	Relative fitness
ab / ab	1
Ab / ab	$1 + 2hs$
aB / ab	$1 + 2hs$
AB / ab	$1 + 2hs$
Ab / Ab	$1 + 2s$
aB / aB	$1 + 2s$
Ab / aB	$1 + 2s$
AB / Ab	$1 + 2s$
AB / aB	$1 + 2s$
AB / AB	$1 + 2s$

Note: Cite [Sanjak]

14.3.2 How do different selection models affect expected LD statistics?

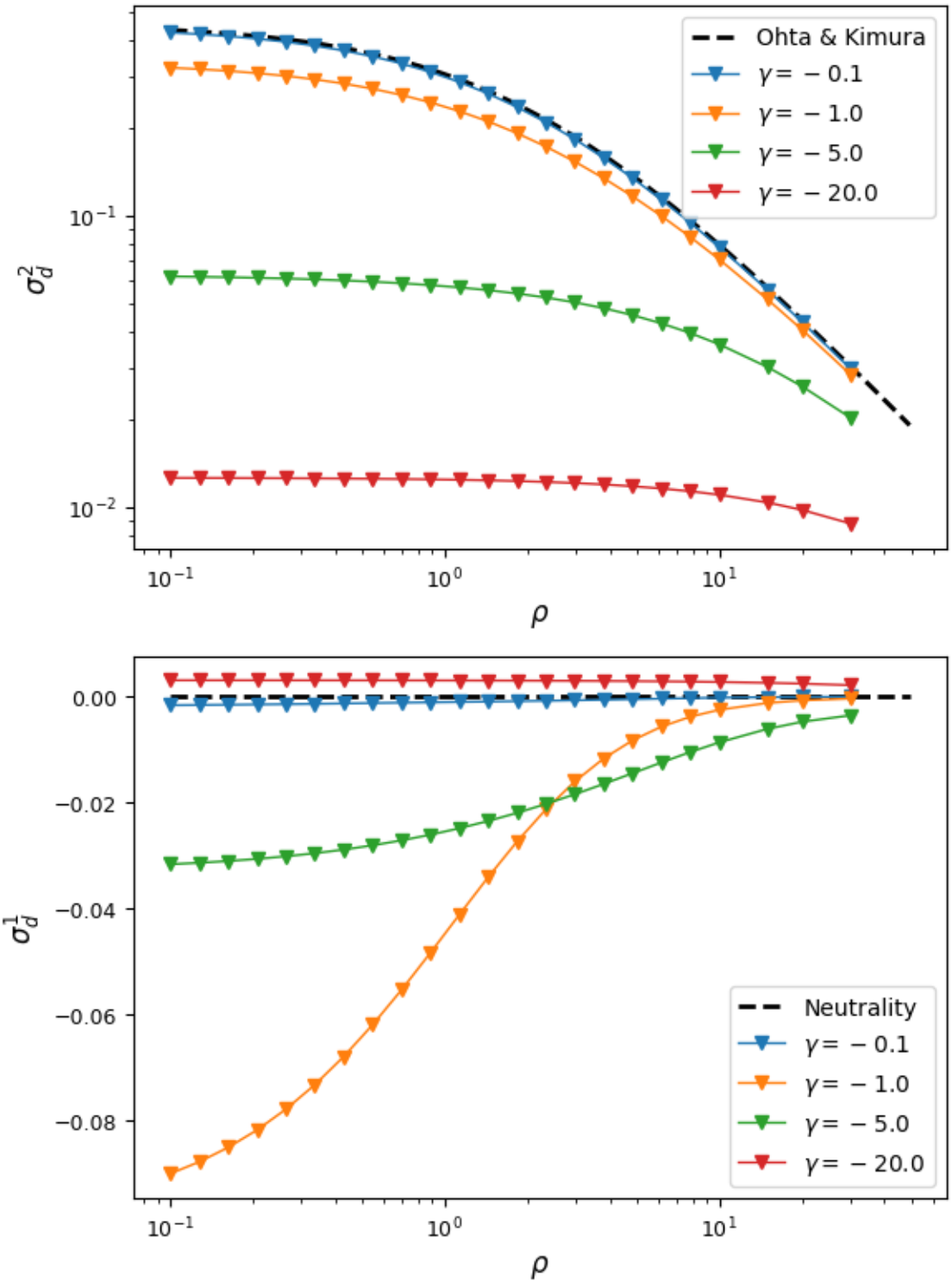
Here, we will examine some relatively simple models in order to gain some intuition about how selection, along with recombination and size changes, affect expected patterns of LD, such as the decay curve of σ_d^2 and Hudson-style slices in the two-locus sampling distribution. The selection coefficients will be equal at the two loci, so that the only selection parameters that change will be the selection models (dominance and epistasis).

Additive selection with and without epistasis

Let's first see how simple, additive selection distorts expected LD away from neutral expectations at steady state. Plotted below are decay curves for both σ_d^2 and $\sigma_d^2 = \mathbb{E}[D]\mathbb{E}[p(1-p)q(1-q)]$, a common signed LD statistic.

For each parameter pair of selection coefficient $\gamma = 2N_e s$ and rho, we use the "helper" function that creates the input selection parameters for the *AB*, *Ab*, and *aB* haplotypes, and then simulate the equilibrium two-locus sampling distribution:

```
sel_params = moments.TwoLocus.Util.additive_epistasis(gamma, epsilon=0)
# epsilon=0 means no epistasis, so s_AB = s_A + s_B
F = moments.TwoLocus.Demographics.equilibrium(n, rho=rho, sel_params=sel_params)
sigma_d1 = F.D() / F.pi2()
sigma_d2 = F.D2() / F.pi2()
```



Already with this very simple selection model (no epistasis, no dominance, equal selection at both loci), we find some interesting behavior. For very strong or very week selection, signed-LD remains close to zero, but for intermediate selection, average D can be significantly negative. As fitness effects get stronger, σ_d^2 is reduced dramatically compare to neutral expectations.

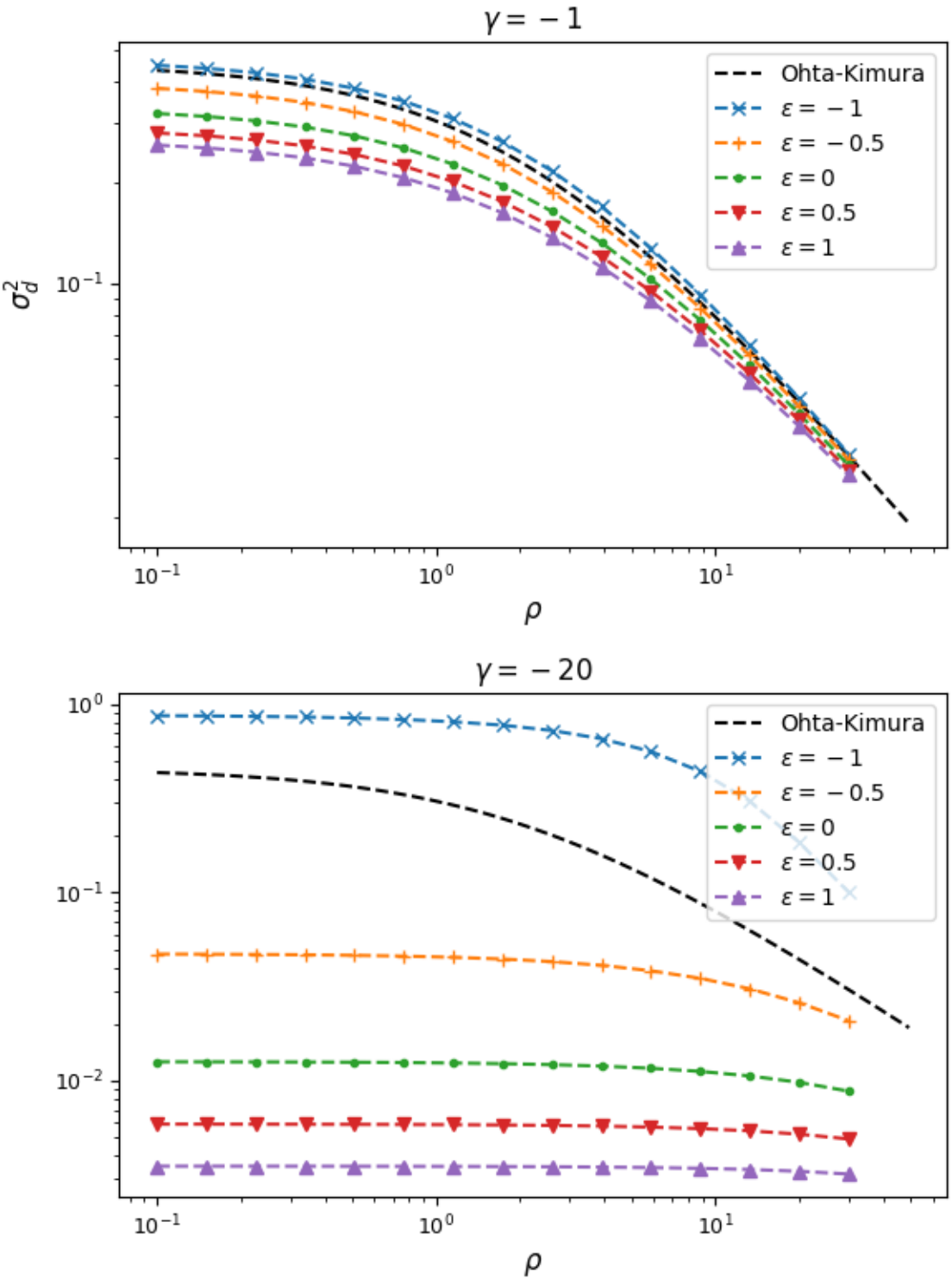
Todo: Plots of frequency conditioned LD.

The "helper" function that we used above converts input γ and ϵ to the selection parameters that are passed to `moments.TwoLocus.Demographics` functions. The additive epistasis model implemented in the helper function (`moments.TwoLocus.Util.additive_epistasis`) returns $[(1+\epsilon)(\gamma_A+\gamma_B), \gamma_A, \gamma_B]$, so that if $\epsilon > 0$, we have synergistic epistasis, and if $\epsilon < 0$, we have antagonistic epistasis. Any value of ϵ is permitted, and note that if ϵ is less than -1, we get reverse-sign epistasis.

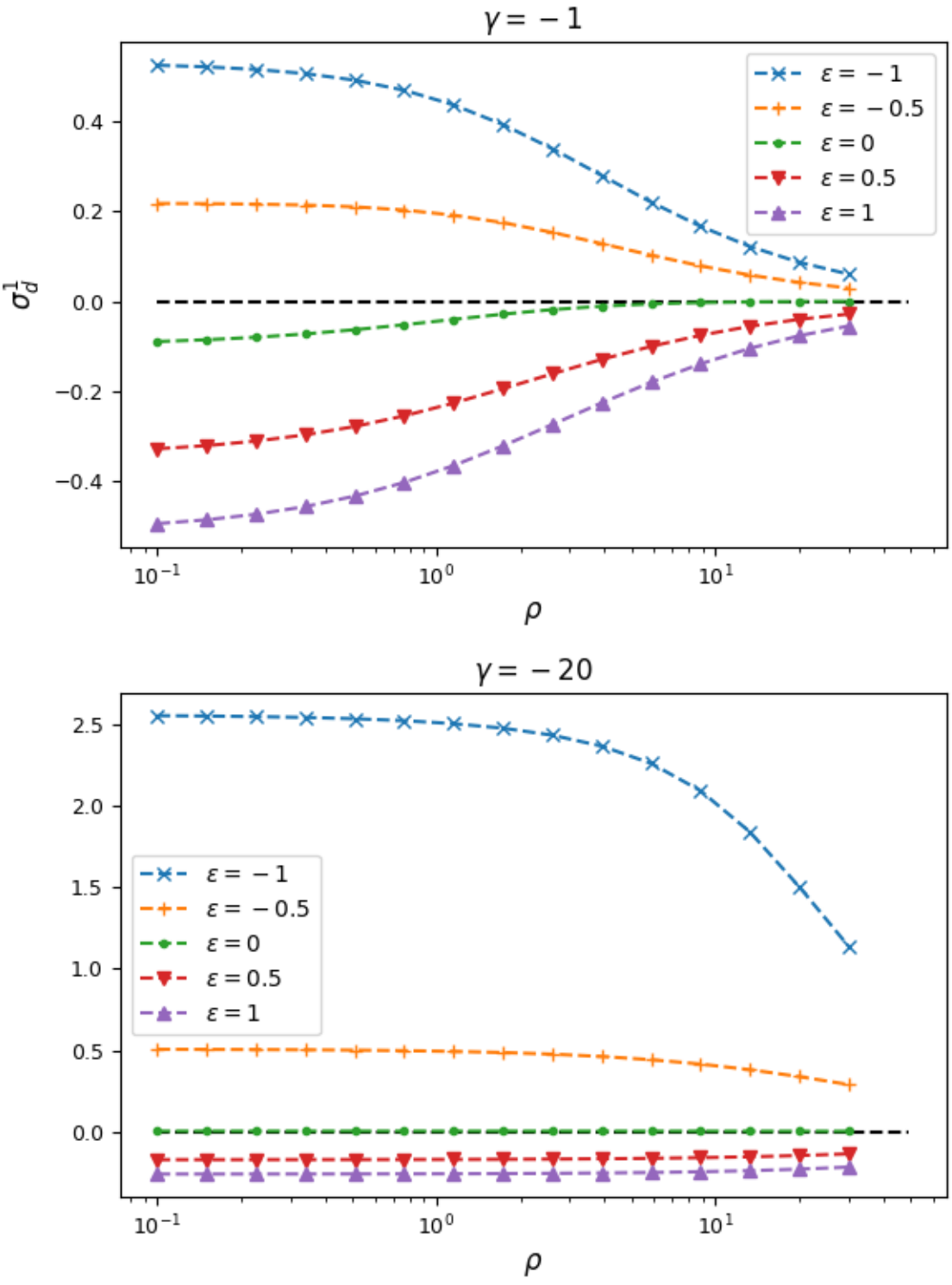
We'll focus on two selection regions: mutations that are slightly deleterious with $\gamma = 1$, and stronger selection with $\gamma = 20$. With an effective population size of 10,000, note that $\gamma = 20$ corresponds to $s = 0.001$ - by no means a lethal mutation, but strong enough to see some interesting differences between selection regimes.

Below we again plot σ_d^2 and σ_d^1 for each set of parameters:

```
gammas = [-1, -20]
epsilons = [-1, -0.5, 0, 0.5, 1]
```



From this, we can see that synergistic epistasis decreases σ_d^2 and antagonistic epistasis increases it above expectations for $\epsilon = 0$. For signed LD, however, both positive and negative ϵ push σ_d^1 farther away from zero:

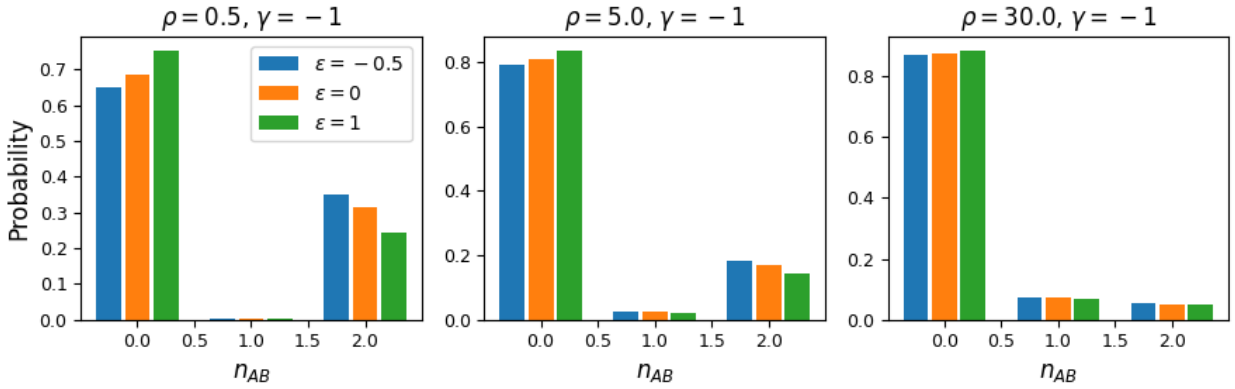


As expected, negative ϵ (i.e. selection against the *AB* haplotype is less strong than the sum of selection against *A* and *B*) leads to an excess of coupling LD (pairs with more *AB* and *ab* haplotypes) than repulsion LD (pairs with more *Ab* and *aB* haplotypes).

We can see this effect more clearly by looking at a slice in the two-locus sampling distribution. Since we're considering negative selection, we'll look at entries in the sampling distribution with low frequencies at the two loci. For doubletons

at both sites:

```
rhos = [0.5, 5.0, 30.0]
n = 30
nA = 2
nB = 2

epsilon = [-0.5, 0, 1]

fig = plt.figure(figsize=(9, 3))
for ii, rho in enumerate(rhos):
    pABs = {}
    for eps in epsilon:
        sel_params = moments.TwoLocus.Util.additive_epistasis(gammas[0], epsilon=eps)
        # F = moments.TwoLocus.Demographics.equilibrium(
        #     n, rho=rho, sel_params=sel_params)
        F = pickle.load(gzip.open(
            f"./data/two-locus/eq.n_{n}.rho_{rho}.sel_"
            + "_".join([str(s) for s in sel_params])
            + ".fs.gz",
            "rb"))
        counts, pAB = moments.TwoLocus.Util.pAB(F, nA, nB)
        pABs[eps] = pAB / pAB.sum()
    ax = plt.subplot(1, 3, ii + 1)
    ax.bar(counts - 0.25, pABs[epsilon[0]], width=0.22, label=rf"$\epsilon={epsilon[0]}$
↪")
    ax.bar(counts, pABs[epsilon[1]], width=0.22, label=rf"$\epsilon={epsilon[1]}$")
    ax.bar(counts + 0.25, pABs[epsilon[2]], width=0.22, label=rf"$\epsilon={epsilon[2]}$
↪")

    ax.set_title(rf"$\rho = {rho}$, $\gamma = {gammas[0]}$")
    ax.set_xlabel(r"$n_{AB}$")
    if ii == 0:
        ax.legend()
        ax.set_ylabel("Probability")
fig.tight_layout()
```

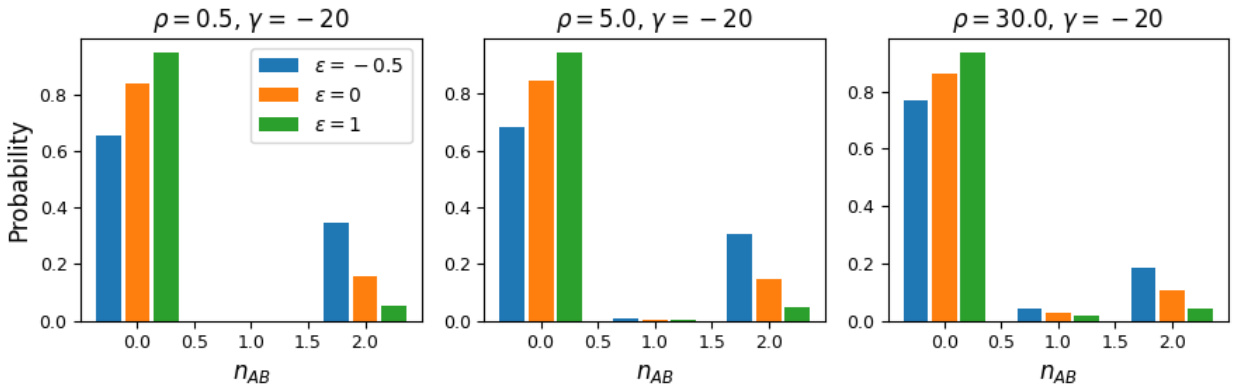



```
fig = plt.figure(figsize=(9, 3))
for ii, rho in enumerate(rhos):
    pABs = {}
```

(continues on next page)

(continued from previous page)

```
    for eps in epsilon:
        sel_params = moments.TwoLocus.Util.additive_epistasis(gammas[1], epsilon=eps)
        # F = moments.TwoLocus.Demographics.equilibrium(
        #     n, rho=rho, sel_params=sel_params)
        F = pickle.load(gzip.open(
            f"./data/two-locus/eq.n_{n}.rho_{rho}.sel_"
            + "_".join([str(s) for s in sel_params])
            + ".fs.gz",
            "rb"))
        counts, pAB = moments.TwoLocus.Util.pAB(F, nA, nB)
        pABs[eps] = pAB / pAB.sum()
    ax = plt.subplot(1, 3, ii + 1)
    ax.bar(counts - 0.25, pABs[epsilon[0]], width=0.22, label=rf"$\epsilon={epsilon[0]}$
↪")
    ax.bar(counts, pABs[epsilon[1]], width=0.22, label=rf"$\epsilon={epsilon[1]}$")
    ax.bar(counts + 0.25, pABs[epsilon[2]], width=0.22, label=rf"$\epsilon={epsilon[2]}$
↪")

    ax.set_title(rf"$\rho = {rho}$, $\gamma = {gammas[1]}$")
    ax.set_xlabel(r"$n_{AB}$")
    if ii == 0:
        ax.legend()
        ax.set_ylabel("Probability")
fig.tight_layout()
```



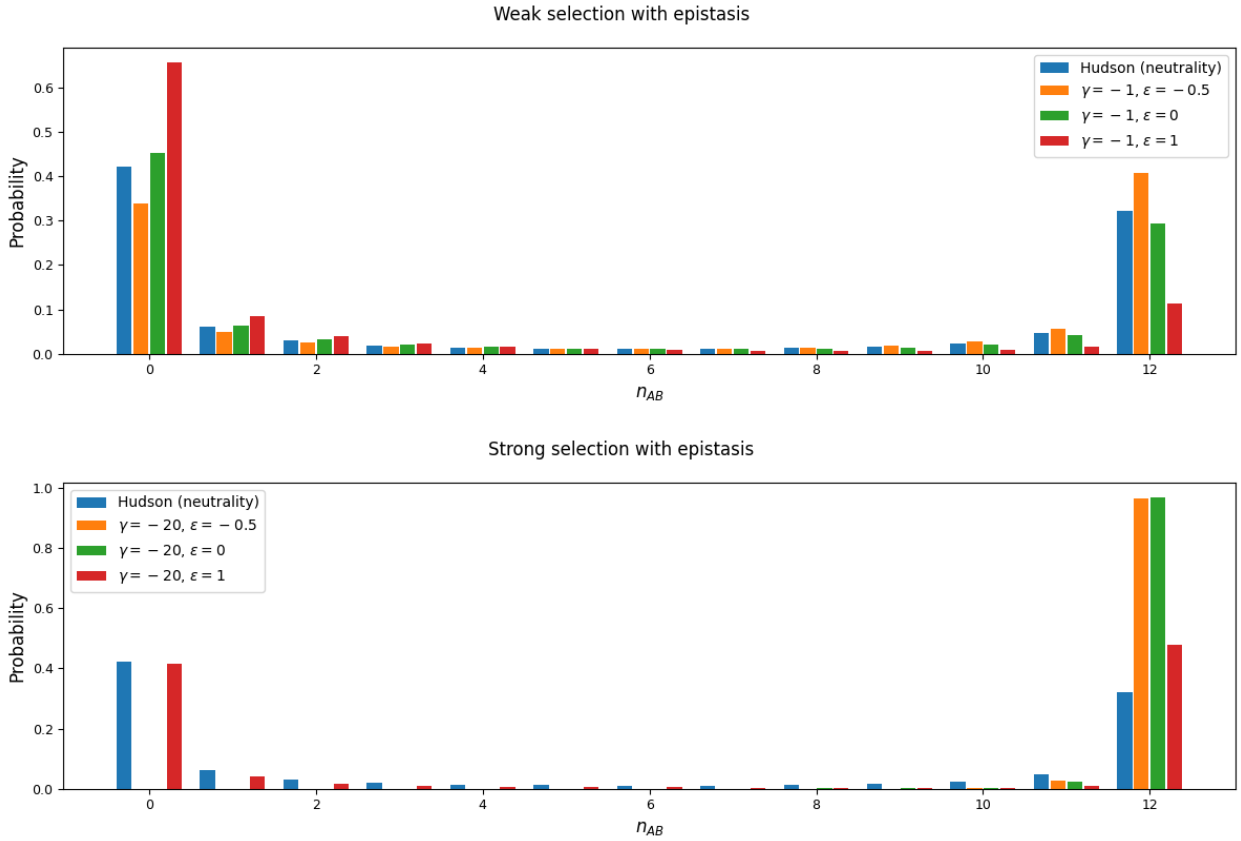
And while very few mutations will reach high frequency, we can also look at the case with $n_A = 15$ and $n_B = 12$ in a sample size of 30. Here, because selection and recombination require the jackknife approximation which works better with larger sample sizes, we solved for the equilibrium distribution using size $n = 60$ and then projected to size 30.

```
n = 60
n_proj = 30
nA = 15
nB = 12
rho = 1

F = moments.TwoLocus.Demographics.equilibrium(n, rho=rho, sel_params=sel_params)
# by default, we usually cache projection steps, but set cache=False here to
# save on memory usage
F_proj = F.project(n_proj, cache=False)
```

(continues on next page)

(continued from previous page)

```
counts, pAB = moments.TwoLocus.Util.pAB(F_proj, nA, nB)
pAB /= pAB.sum()
```



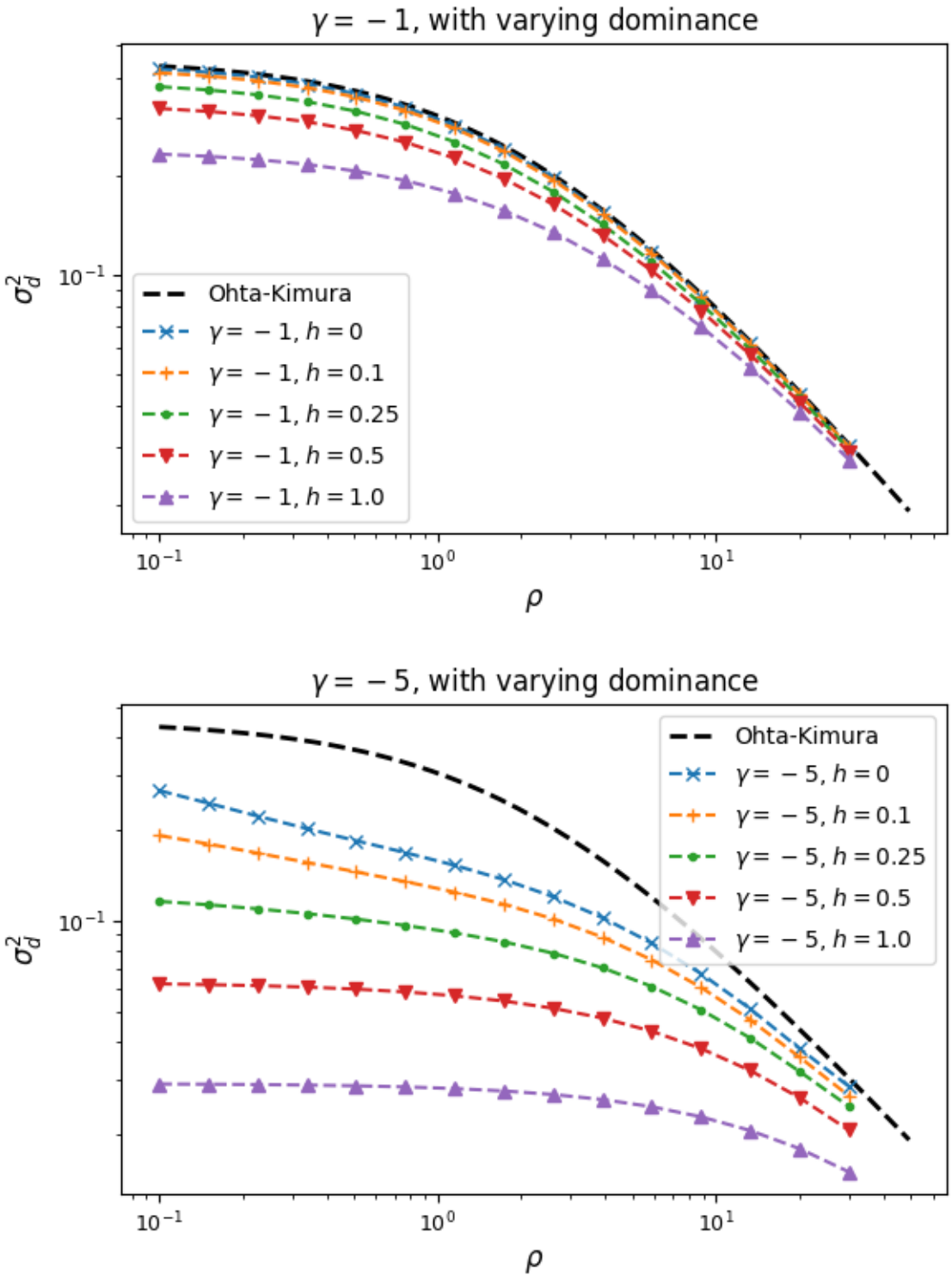
Dominance

We again assume fitness effects are the same at both loci, and now explore how dominance affects LD. We'll start by looking at the "simple" dominance model without epistasis, so that fitness effects are additive across loci. When simulating with dominance, the selection model no longer collapses to a haploid model, but instead we need to specify the selection coefficients for each possible diploid haplotype pair *AB/AB*, *AB/Ab*, etc. We'll use another helper function to generate those selection coefficients and pass them to the `sel_params_general` keyword argument.

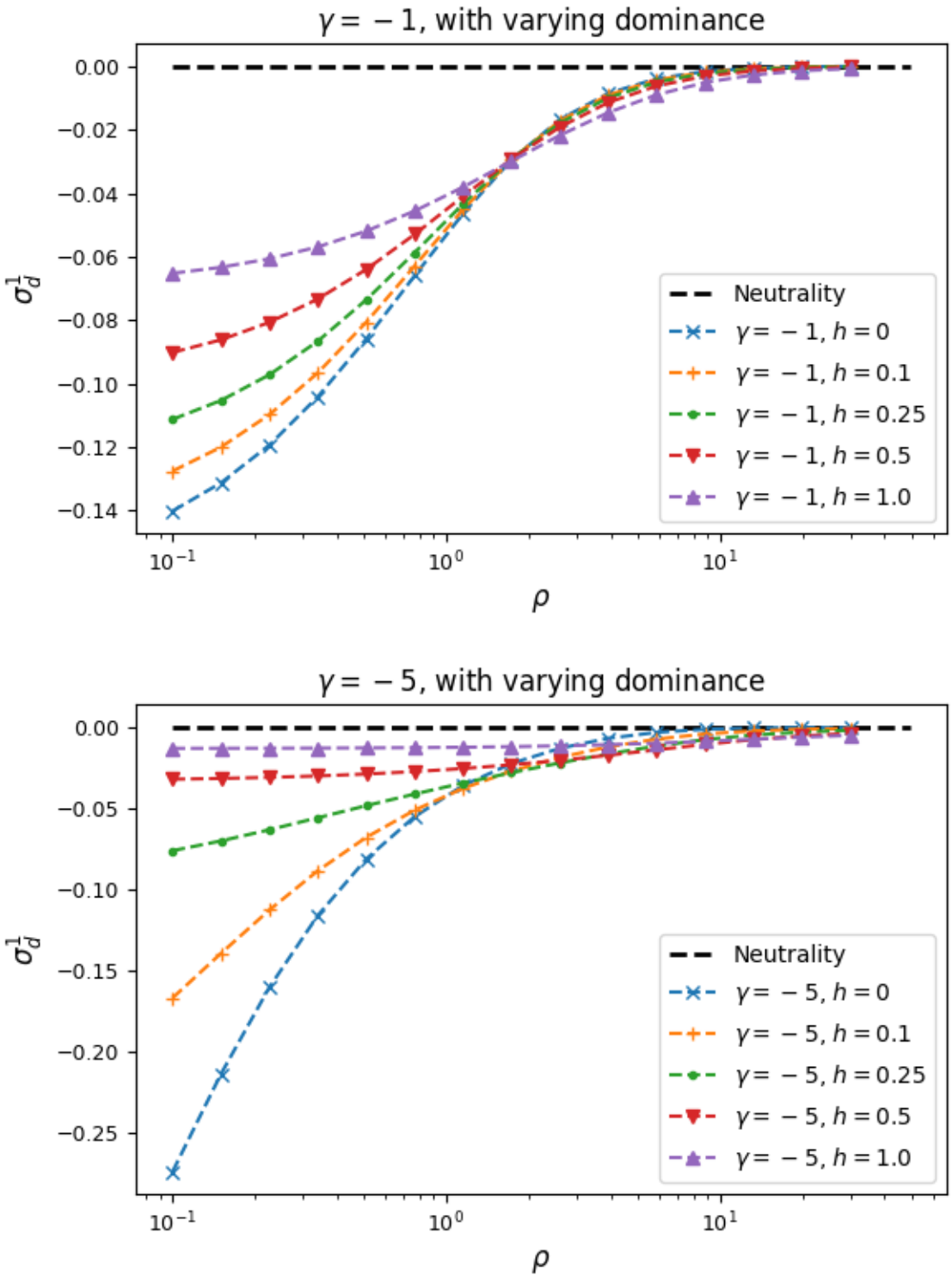
For example, to simulate the equilibrium distribution with selection coefficient -5 and dominance coefficient 0.1 under the simple dominance model:

```
gamma = -5
h = 0.1
sel_params = moments.TwoLocus.Util.simple_dominance(gamma, h=h)
F = moments.TwoLocus.Demographics.equilibrium(n, rho, sel_params_general=sel_params)
```

Let's look at how σ_d^2 and σ_d^1 are affected by dominance.



Squared LD (σ_d^2) is increased for recessive variants, while pairs of dominant mutations reduce it below expectations for additive variants.



Similarly, recessive mutations lead to larger average negative signed LD. However, this pattern also depends on the underlying selection coefficient, with LD decay curves that can vary qualitatively for different selection coefficients and recombination rates between loci, even when dominance is equivalent.

Todo: Relate to associative overdominance work, e.g. Charlesworth, and Hill-Robertson interference.

Gene-based dominance

Todo: All the comparisons, show LD curves and expectations for signed LD, depending on the selection model, maybe explore how population size changes distort these expectations.

Non-steady-state demography

$\mathcal{K}$

Todo: Are any of these statistics quite sensitive to bottlenecks or expansions?

Todo: Discussion on what we can expect to learn from signed LD-based inferences. Are the various selection models and demography hopelessly confounded?

14.4 References

CHAPTER

FIFTEEN

API FOR SITE FREQUENCY SPECTRA

15.1 The Spectrum object

class moments.**Spectrum**(*data*, *mask=False*, *mask_corners=True*, *data_folded=None*, *check_folding=True*, *dtype=<class 'float'>*, *copy=True*, *fill_value=nan*, *keep_mask=True*, *shrink=True*, *pop_ids=None*)

Represents a single-locus biallelic frequency spectrum.

Spectra are represented by masked arrays. The masking allows us to ignore specific entries in the spectrum. When simulating under the standard infinite sites model (ISM), the entries we mask are the bins specifying absent or fixed variants. When using a reversible mutation model (i.e. the finite genome model), we track the density of variants in fixed bins, setting `mask_corners` to `False`.

Parameters

- **data** (*array*) – An array with dimension equal to the number of populations. Each dimension has length $n_i + 1$, where n_i is the sample size for the i-th population.
- **mask** – An optional array of the same size as data. 'True' entries in this array are masked in the Spectrum. These represent missing data categories. (For example, you may not trust your singleton SNP calling.)
- **mask_corners** – If True (default), the 'observed in none' and 'observed in all' entries of the FS will be masked. Typically these entries are masked. In the defaul infinite sites model, moments does not reliably calculate the fixed-bin entries, so you will almost always want `mask_corners=True`. The exception is if we are simulating under the finite genome model, in which case we track the probability of a site to be fixed for either allele.
- **data_folded** (*bool, optional*) – If True, it is assumed that the input data is folded. An error will be raised if the input data and mask are not consistent with a folded Spectrum.
- **check_folding** (*bool, optional*) – If True and data_folded=True, the data and mask will be checked to ensure they are consistent with a folded Spectrum. If they are not, a warning will be printed.
- **pop_ids** (*list of strings, optional*) – Optional list of strings containing the population labels, with length equal to the dimension of `data`.

Returns

A frequency spectrum object, as a masked array.

Fst(*pairwise=False*)

Wright's Fst between the populations represented in the fs.

This estimate of Fst assumes random mating, because we don't have heterozygote frequencies in the fs.

Calculation is by the method of Weir and Cockerham *Evolution* 38:1358 (1984). For a single SNP, the relevant formula is at the top of page 1363. To combine results between SNPs, we use the weighted average indicated by equation 10.

Parameters

pairwise (*bool*) – Defaults to False. If True, returns a dictionary of all pairwise Fst within the multi-dimensional spectrum.

`S()`

Returns the number of segregating sites in the frequency spectrum.

`Tajima_D()`

Returns Tajima's D.

Following Gillespie "Population Genetics: A Concise Guide" pg. 45

`Watterson_theta()`

Returns Watterson's estimator of theta.

Note: This function is only sensible for 1-dimensional spectra.

`Zengs_E()`

Returns Zeng et al.'s E statistic.

From Zeng et al., "Statistical Tests for Detecting Positive Selection by Utilizing High-Frequency Variants." *Genetics*, 2016.

`admix(idx0, idx1, num_lineages, proportion, new_id=None)`

Returns a new frequency spectrum with an admixed population that arose through admixture from indexed populations with given number of lineages and proportions from parental populations. This serves as a wrapper for `Manips.admix_into_new`, with the added feature of handling pop_ids.

If the number of lineages that move are equal to the number of lineages previously present in a source population, that source population is marginalized.

Parameters

- **idx0** (*int*) – Index of first source population.
- **idx1** (*int*) – Index of second source population.
- **num_lineages** (*int*) – Number of lineages in the new population. Cannot be greater than the number of existing lineages in either source populations.
- **proportion** (*float*) – The proportion of lineages that come from the first source population (1-proportion acestry comes from the second source population). Must be a number between 0 and 1.
- **new_id** (*str, optional*) – The ID of the new population. Can only be used if the population IDs are specified in the input SFS.

`branch(idx, n, new_id=None)`

A "branch" event, where a population gives rise to a child population, while persisting. This is conceptually similar to the split event. The number of lineages in the new population is provided, and the number of lineages in the source/parental population is the original sample size minus the number requested for the branched population. Returns a new frequency spectrum.

Parameters

- **idx** (*int*) – The index of the population to branch.

- **n** (*int*) – The sample size of the new population.
- **new_id** – The population ID of the branch populations. The parental population retains its original population ID. Can only be used if pop_ids are given for the input spectrum.

fixed_size_sample(*nsamples*, *include_masked=False*)

Generate a resampled SFS from the current one. Thus, the resampled SFS follows a multinomial distribution given by the proportion of sites in each bin in the original SFS.

Parameters

- **nsamples** (*int*) – Number of samples to include in the new SFS.
- **include_masked** (*bool, optional*) – If True, use all bins from the SFS. Otherwise, use only non-masked bins. Defaults to False.

fold()

Returns a folded frequency spectrum.

The folded fs assumes that information on which allele is ancestral or derived is unavailable. Thus the fs is in terms of minor allele frequency. This makes the fs into a "triangular" array. If a masked cell is folded into non-masked cell, the destination cell is masked as well.

Folding is not done in-place. The return value is a new Spectrum object.

static from_angsd(*sfs_file*, *sample_sizes*, *pop_ids=None*, *folded=False*, *mask_corners=True*)

Convert ANGSD output to a moments Spectrum object. The sample sizes are given as number of haploid genome copies (twice the number of sampled diploid individuals).

Parameters

- **sfs_file** (*string*) – The n-dimensional SFS from ANGSD. This should be a file with a single line of numbers, as entries in the SFS.
- **sample_sizes** (*list*) – A list of integers with length equal to the number of population, storing the haploid sample size in each population. The order must match the population order provided to ANGSD.
- **pop_ids** (*list*) – A list of strings equal with length equal to the number of population, specifying the population name for each.
- **folded** (*bool*) – If False (default), we assume ancestral states are known, returning an unfolded SFS. If True, the returned SFS is folded.
- **mask_corners** (*bool*) – If True (default), mask the fixed bins in the SFS. If False, the fixed bins will remain unmasked.

Returns

A `moments` site frequency spectrum.

Return type

moments.Spectrum

static from_data_dict(*data_dict*, *pop_ids*, *projections*, *mask_corners=True*, *polarized=True*)

Spectrum from a dictionary of polymorphisms.

The data dictionary should be organized as:

```
{snp_id: {
    'segregating': ['A','T'],
    'calls': {
        'YRI': (23,3),
```

(continues on next page)

(continued from previous page)

```
        'CEU': (7,3)
    },
    'outgroup_allele': 'T'
}}
```

The 'calls' entry gives the successful calls in each population, in the order that the alleles are specified in 'segregating'. Non-diallelic polymorphisms are skipped.

Parameters

- **pop_ids** – list of which populations to make fs for.
- **projections** – list of sample sizes to project down to for each population.
- **polarized** – If True, the data are assumed to be correctly polarized by 'outgroup_allele'. SNPs in which the 'outgroup_allele' information is missing or '-' or not concordant with the segregating alleles will be ignored. If False, any 'outgroup_allele' info present is ignored, and the returned spectrum is folded.

static from_demes(*g*, *sampled_demes=None*, *sample_sizes=None*, *sample_times=None*, *samples=None*, *Ne=None*, *unsampled_n=4*, *gamma=None*, *h=None*)

Takes a deme graph and computes the SFS. `demes` is a package for specifying demographic models in a user-friendly, human-readable YAML format. This function automatically parses the demographic description and returns a SFS for the specified populations and sample sizes.

Note: If a deme sample time is requested that is earlier than the deme's end time, for example to simulate ancient samples, we must create a new population for that ancient sample. This can cause large slow-downs, as the computation cost of computing the SFS grows quickly in the number of populations.

Parameters

- **g** (str or `demes.DemeGraph`) – A `demes` DemeGraph from which to compute the SFS. The DemeGraph can either be specified as a YAML file, in which case *g* is a string, or as a `DemeGraph` object.
- **sampled_demes** (*list of strings*) – A list of deme IDs to take samples from. We can repeat demes, as long as the sampling of repeated deme IDs occurs at distinct times.
- **sample_sizes** (*list of ints*) – A list of the same length as `sampled_demes`, giving the sample sizes for each sampled deme.
- **sample_times** (*list of floats, optional*) – If None, assumes all sampling occurs at the end of the existence of the sampled deme. If there are ancient samples, `sample_times` must be a list of same length as `sampled_demes`, giving the sampling times for each sampled deme. Sampling times are given in time units of the original deme graph, so might not necessarily be generations (e.g. if `g.time_units` is years)
- **Ne** (*float, optional*) – reference population size. If none is given, we use the initial size of the root deme.
- **unsampled_n** (*int, optional*) – The default sample size of unsampled demes, which must be greater than or equal to 4.
- **gamma** (*float or dict*) – The scaled selection coefficient(s), 2*Ne*s. Defaults to None, which implies neutrality. Can be given as a scalar value, in which case all populations have the same selection coefficient. Alternatively, can be given as a dictionary, with keys given as population names in the input Demes model. Any population missing from this dictionary

will be assigned a selection coefficient of zero. A non-zero default selection coefficient can be provided, using the key *_default*. See the Demes exension documentation for more details and examples.

- **h** (*float or dict*) – The dominance coefficient(s). Defaults to additivity (or genic selection). Can be given as a scalar value, in which case all populations have the same dominance coefficient. Alternatively, can be given as a dictionary, with keys given as population names in the input Demes model. Any population missing from this dictionary will be assigned a dominance coefficient of 1/2 (additivity). A different default dominance coefficient can be provided, using the key *_default*. See the Demes exension documentation for more details and examples.

Returns

A `moments` site frequency spectrum, with dimension equal to the length of `sampled_demes`, and shape equal to `sample_sizes` plus one in each dimension, indexing the allele frequency in each deme from 0 to n[i], where i is the deme index.

Return type

moments.Spectrum

static from_file(*fid*, *mask_corners=True*, *return_comments=False*)

Read frequency spectrum from file.

See `to_file` for details on the file format.

Parameters

- **fid** (*string*) – string with file name to read from or an open file object.
- **mask_corners** (*bool, optional*) – If True, mask the 'absent in all samples' and 'fixed in all samples' entries.
- **return_comments** (*bool, optional*) – If true, the return value is (fs, comments), where comments is a list of strings containing the comments from the file (without #'s).

static from_ms_file(*fid*, *average=True*, *mask_corners=True*, *return_header=False*, *pop_assignments=None*, *pop_ids=None*, *bootstrap_segments=1*)

Read frequency spectrum from file of ms output.

Parameters

- **fid** – string with file name to read from or an open file object.
- **average** – If True, the returned fs is the average over the runs in the ms file. If False, the returned fs is the sum.
- **mask_corners** – If True, mask the 'absent in all samples' and 'fixed in all samples' entries.
- **return_header** – If True, the return value is (fs, (command,seeds), where command and seeds are strings containing the ms commandline and the seeds used.
- **pop_assignments** – If None, the assignments of samples to populations is done automatically, using the assignment in the ms command line. To manually assign populations, pass a list of the from [6,8]. This example places the first 6 samples into population 1, and the next 8 into population 2.
- **pop_ids** – Optional list of strings containing the population labels. If pop_ids is None, labels will be "pop0", "pop1", ...
- **bootstrap_segments** – If bootstrap_segments is an integer greater than 1, the data will be broken up into that many segments based on SNP position. Instead of single FS, a list of spectra will be returned, one for each segment.

static fromfile(*fid*, *mask_corners=True*, *return_comments=False*)

Read frequency spectrum from file.

See `to_file` for details on the file format.

Parameters

- **fid** (*string*) – string with file name to read from or an open file object.
- **mask_corners** (*bool, optional*) – If True, mask the 'absent in all samples' and 'fixed in all samples' entries.
- **return_comments** (*bool, optional*) – If true, the return value is (fs, comments), where comments is a list of strings containing the comments from the file (without #'s).

genotype_matrix(*num_sites=None*, *sample_sizes=None*, *diploid_genotypes=False*)

Generate a genotype matrix of independent loci. For multi-population spectra, the individual columns are filled in the sample order as the populations in the SFS.

Note: Sites in the output genotype matrix are necessarily separated by infinite recombination. The SFS assumes all loci are segregating independently, so there is no linkage between them.

Returns a genotype matrix of size number of sites by total sample size.

Parameters

- **num_sites** – Defaults to None, in which case we take a poisson sample from the SFS. Otherwise, we take a fixed number of sites.
- **sample_sizes** – The sample size in each population, as a list with length of the number of dimension (populations) in the SFS.

Diploid_genotypes

Defaults to False, in which case we return a haplotype matrix of size (num_sites x sum(sample_sizes)). If True, we return a diploid genotype matrix (filled with 0, 1, 2) of size (num_sites x sum(sample_sizes)/2).

integrate(*Npop*, *tf*, *dt_fac=0.02*, *gamma=None*, *h=None*, *m=None*, *theta=1.0*, *adapt_dt=False*, *finite_genome=False*, *theta_fd=None*, *theta_bd=None*, *frozen=[False]*)

Method to simulate the spectrum's evolution for a given set of demographic parameters. The SFS is integrated forward-in-time, and the integration occurs in-place, meaning you need only call `fs.integrate()`, and the `fs` is updated.

Parameters

- **Npop** (*list or function that returns a list*) – List of populations' relative effective sizes. Can be given as a list of positive values for constant sizes, or as a function that returns a list of sizes at a given time.
- **tf** (*float*) – The total integration time in genetic units.
- **dt_fac** (*float, optional*) – The timestep factor, default is 0.02. This parameter typically does not need to be adjusted.
- **gamma** (*float or list of floats, optional*) – The selection coefficient ($2N_e s$), or list of selection coefficients if more than one population.
- **h** (*float or list of floats, optional*) – The dominance coefficient, or list of dominance coefficients in each population, if more than one population.

- **m** (*array-like, optional*) – The migration rates matrix as a 2-D array with shape nxn, where n is the number of populations. The entry of the migration matrix m[i,j] is the migration rate from pop j to pop i in genetic units, that is, normalized by $2N_e$. *m* may be either a 2-D array, or a function that returns a 2-D array (with dimensions equal to (num pops)x(num pops)).
- **theta** (*float, optional*) – The scaled mutation rate $4N_e u$, which defaults to 1. `theta` can be used in the reversible model in the case of symmetric mutation rates. In this case, `theta` must be set to << 1.
- **adapt_dt** (*bool, optional*) – flag to allow dt correction avoiding negative entries.
- **finite_genome** (*bool, optional*) – If True, simulate under the finite-genome model with reversible mutations. If using this model, we can specify the forward and backward mutation rates, which are per-base rates that are not scaled by number of mutable loci. If `theta_fd` and `theta_bd` are not specified, we assume equal forward and backward mutation rates provided by `theta`, which must be set to less that 1. Defaults to False.
- **theta_fd** (*float, optional*) – The forward mutation rate $4Neu$.
- **theta_bd** (*float, optional*) – The backward mutation rate $4Nev$.
- **frozen** (*list of bools*) – Specifies the populations that are "frozen", meaning samples from that population no longer change due or contribute to migration to other populations. This feature is most often used to indicate ancient samples, for example, ancient DNA. The `frozen` parameter is given as a list of same length as number of pops, with `True` for frozen populations at the corresponding index, and `False` for populations that continue to evolve.

log()

Returns the natural logarithm of the entries of the frequency spectrum.

Only necessary because np.ma.log now fails to propagate extra attributes after np 1.10.

marginalize(*over*, *mask_corners=None*)

Reduced dimensionality spectrum summing over the set of populations given by `over`.

`marginalize` does not act in-place, so the input frequency spectrum will not be altered.

Parameters

- **over** (*list of integers*) – List of axes to sum over. For example (0,2) will marginalize populations 0 and 2.
- **mask_corners** (*bool, optional*) – If True, the fixed bins of the resulting spectrum will be masked. The default behavior is to mask the corners only if at least one of the corners of the input frequency spectrum is masked. If either corner is masked, the output frequency spectrum masks the fixed bins.

mask_corners()

Mask the 'seen in 0 samples' and 'seen in all samples' entries.

pi()

Returns the estimated expected number of pairwise differences between two chromosomes in the population.

Note: This estimate includes a factor of sample_size / (sample_size - 1) to make $\mathbb{E}[\pi] = \theta$.

---

project(*ns*)

Project to smaller sample size.

project does *not* act in-place, so that the input frequency spectrum is not changed.

Parameters
ns (*list of integers*) – Sample sizes for new spectrum.

pulse_migrate(*idx_from*, *idx_to*, *keep_from*, *proportion*)

Mass migration (pulse admixture) between two existing populations. The target (destination) population has the same number of lineages in the output SFS, and the source population has keep_from number of lineages after the pulse event. The proportion is the expected ancestry proportion in the target population that comes from the source population.

This serves as a wrapper for Manips.admix_inplace.

Depending on the proportion and number of lineages, because this is an approximate operation, we often need a large number of lineages from the source population to maintain accuracy.

Parameters

- **idx_from** (*int*) – Index of source population.
- **idx_to** (*int*) – Index of targeet population.
- **keep_from** (*int*) – Number of lineages to keep in source population.
- **proportion** (*float*) – Ancestry proportion of source population that migrates to target population.

sample()

Generate a Poisson-sampled fs from the current one.

Entries where the current fs is masked or 0 will be masked in the output sampled fs.

scramble_pop_ids(*mask_corners=True*)

Spectrum corresponding to scrambling individuals among populations.

This is useful for assessing how diverged populations are. Essentially, it pools all the individuals represented in the fs and generates new populations of random individuals (without replacement) from that pool. If this fs is significantly different from the original, that implies population structure.

split(*idx*, *n0*, *n1*, *new_ids=None*)

Splits a population in the SFS into two populations, with the extra population placed at the end. Returns a new frequency spectrum.

Parameters

- **idx** (*int*) – The index of the population to split.
- **n0** (*int*) – The sample size of the first split population.
- **n1** (*int*) – The sample size of the second split population.
- **new_ids** (*list of strings, optional*) – The population IDs of the split populations. Can only be used if pop_ids are given for the input spectrum.

swap_axes(*ax1*, *ax2*)

Uses np's swapaxes function, but also swaps pop_ids as appropriate if pop_ids are given.

Note: fs.swapaxes(ax1, ax2) will still work, but if population ids are given, it won't swap the pop_ids entries as expected.

Parameters

- **ax1** (*int*) – The index of the first population to swap.
- **ax2** (*int*) – The index of the second population to swap.

theta_L()

Returns theta_L as defined by Zeng et al. "Statistical Tests for Detecting Positive Selection by Utilizing High-Frequency Variants" (2006) Genetics

---

Note: This function is only sensible for 1-dimensional spectra.

---

to_file(*fid*, *precision=16*, *comment_lines=[]*, *foldmaskinfo=True*)

Write frequency spectrum to file.

The file format is:

- Any number of comment lines beginning with a '#'
- A single line containing N integers giving the dimensions of the fs array. So this line would be '5 5 3' for an SFS that was 5x5x3. (That would be 4x4x2 *samples*.)
- On the *same line*, the string 'folded' or 'unfolded' denoting the folding status of the array
- On the *same line*, optional strings each containing the population labels in quotes separated by spaces, e.g. "pop 1" "pop 2"
- A single line giving the array elements. The order of elements is e.g.: fs[0,0,0] fs[0,0,1] fs[0,0,2] ... fs[0,1,0] fs[0,1,1] ...
- A single line giving the elements of the mask in the same order as the data line. '1' indicates masked, '0' indicates unmasked.

Parameters

- **fid** (*string*) – string with file name to write to or an open file object.
- **precision** (*int, optional*) – precision with which to write out entries of the SFS. (They are formated via %.<p>g, where <p> is the precision.) Defaults to 16.
- **comment_lines** (*list of strings, optional*) – list of strings to be used as comment lines in the header of the output file.
- **foldmaskinfo** (*bool, optional*) – If False, folding and mask and population label information will not be saved.

tofile(*fid*, *precision=16*, *comment_lines=[]*, *foldmaskinfo=True*)

Write frequency spectrum to file.

The file format is:

- Any number of comment lines beginning with a '#'
- A single line containing N integers giving the dimensions of the fs array. So this line would be '5 5 3' for an SFS that was 5x5x3. (That would be 4x4x2 *samples*.)
- On the *same line*, the string 'folded' or 'unfolded' denoting the folding status of the array
- On the *same line*, optional strings each containing the population labels in quotes separated by spaces, e.g. "pop 1" "pop 2"

- A single line giving the array elements. The order of elements is e.g.: fs[0,0,0] fs[0,0,1] fs[0,0,2] ... fs[0,1,0] fs[0,1,1] ...
- A single line giving the elements of the mask in the same order as the data line. '1' indicates masked, '0' indicates unmasked.

Parameters

- **fid** (*string*) – string with file name to write to or an open file object.
- **precision** (*int, optional*) – precision with which to write out entries of the SFS. (They are formated via %.<p>g, where <p> is the precision.) Defaults to 16.
- **comment_lines** (*list of strings, optional*) – list of strings to be used as comment lines in the header of the output file.
- **foldmaskinfo** (*bool, optional*) – If False, folding and mask and population label information will not be saved.

unfold()

Returns an unfolded frequency spectrum.

It is assumed that each state of a SNP is equally likely to be ancestral.

Unfolding is not done in-place. The return value is a new Spectrum object.

unmask_all()

Unmask all entires of the frequency spectrum.

15.2 Miscellaneous functions

moments.Misc.**perturb_params**(*params*, *fold=1*, *lower_bound=None*, *upper_bound=None*)

Generate a perturbed set of parameters. Each element of params is randomly perturbed *fold* factors of 2 up or down.

Parameters

- **fold** (*float, optional*) – Number of factors of 2 to perturb by, defaults to 1.
- **lower_bound** (*list of floats, optional*) – If not None, the resulting parameter set is adjusted to have all value greater than lower_bound.
- **upper_bound** (*list of floats, optional*) – If not None, the resulting parameter set is adjusted to have all value less than upper_bound.

moments.Misc.**make_data_dict_vcf**(*vcf_filename*, *popinfo_filename*, *filter=True*, *flanking_info=[None, None]*, *skip_multiallelic=True*)

Parse a VCF file containing genomic sequence information, along with a file identifying the population of each sample, and store the information in a properly formatted dictionary.

Each file may be zipped (.zip) or gzipped (.gz). If a file is zipped, it must be the only file in the archive, and the two files cannot be zipped together. Both files must be present for the function to work.

Parameters

- **vcf_filename** (*str*) – Name of VCF file to work with. The function currently works for biallelic SNPs only, so if REF or ALT is anything other than a single base pair (A, C, T, or G), the allele will be skipped. Additionally, genotype information must be present in the FORMAT field GT, and genotype info must be known for every sample, else the SNP will be

skipped. If the ancestral allele is known it should be specified in INFO field 'AA'. Otherwise, it will be set to '-'.

- **popinfo_filename** (*str*) – Name of file containing the population assignments for each sample in the VCF. If a sample in the VCF file does not have a corresponding entry in this file, it will be skipped. See _get_popinfo for information on how this file must be formatted.
- **filter** (*bool, optional*) – If set to True, alleles will be skipped if they have not passed all filters (i.e. either 'PASS' or '.' must be present in FILTER column.
- **flanking_info** (*list of strings, optional*) – Flanking information for the reference and/or ancestral allele can be provided as field(s) in the INFO column. To add this information to the dict, flanking_info should specify the names of the fields that contain this info as a list (e.g. ['RFL', 'AFL'].) If context info is given for only one allele, set the other item in the list to None, (e.g. ['RFL', None]). Information can be provided as a 3 base-pair sequence or 2 base-pair sequence, where the first base-pair is the one immediately preceding the SNP, and the last base-pair is the one immediately following the SNP.
- **skip_multiallelic** (*bool, optional*) – If True, only keep biallelic sites, and skip sites that have more than one ALT allele.

moments.Misc.**count_data_dict**(*data_dict*, *pop_ids*)

Summarize data in data_dict by mapping SNP configurations to counts.

Returns a dictionary with keys (successful_calls, derived_calls, polarized) mapping to counts of SNPs. Here successful_calls is a tuple with the number of good calls per population, derived_calls is a tuple of derived calls per pop, and polarized indicates whether that SNP was polarized using an ancestral state.

Parameters

- **data_dict** (*data dictionary*) – data_dict formatted as in Misc.make_data_dict
- **pop_ids** (*list of strings*) – IDs of populations to collect data for

moments.Misc.**bootstrap**(*data_dict*, *pop_ids*, *projections*, *mask_corners=True*, *polarized=True*, *bed_filename=None*, *num_boots=100*, *save_dir=None*)

Use a non-parametric bootstrap on SNP information contained in a dictionary to generate new data sets. The new data is created by sampling with replacement from independent units of the original data. These units can simply be chromosomes, or they can be regions specified in a BED file.

This function either returns a list of all the newly created SFS, or writes them to disk in a specified directory.

See *moments.Spectrum.from_data_dict()* for more details about the options for creating spectra.

Parameters

- **data_dict** (*dict of SNP information*) – Dictionary containing properly formatted SNP information (i.e. created using one of the make_data_dict methods).
- **pop_ids** (*list of strings*) – List of population IDs.
- **projections** (*list of ints*) – Projection sizes for the given population IDs.
- **mask_corners** (*bool, optional*) – If True, mask the invariant bins of the SFS.
- **polarized** (*bool, optional*) – If True, we assume we know the ancestral allele. If False, return folded spectra.
- **bed_filename** (*string as path to bed file*) – If None, chromosomes will be used as the units for resampling. Otherwise, this should be the filename of a BED file specifying the regions to be used as resampling units. Chromosome names must be consistent between the BED file and the data dictionary, or bootstrap will not work. For example, if an entry in the data dict has ID X_Y, then the value in in the chromosome field of the BED file must

also be X (not chrX, chromosomeX, etc.). If the name field is provided in the BED file, then any regions with the same name will be considered to be part of the same unit. This may be useful for sampling as one unit a gene that is located across non-continuous regions.

- **num_boots** (*int, optional*) – Number of resampled SFS to generate.
- **save_dir** (*str, optional*) – If None, the SFS are returned as a list. Otherwise this should be a string specifying the name of a new directory under which all of the new SFS should be saved.

15.3 Demographic functions

Single-population demographic models.

moments.Demographics1D.**bottlegrowth**(*params*, *ns*, *pop_ids=None*)

Instantanous size change followed by exponential growth.

params = (nuB, nuF, T)

Parameters

- **params** – Tuple of length three specifying (nuB, nuF, T).
  - nuB: Ratio of population size after instantanous change to ancient population size.
  - nuF: Ratio of contemporary to ancient population size.
  - T: Time in the past at which instantaneous change happened and growth began (in units of 2*Na generations).
- **ns** – Number of samples in resulting Spectrum.
- **pop_ids** – Optional list of length one specifying the population ID.

moments.Demographics1D.**growth**(*params*, *ns*, *pop_ids=None*)

Exponential growth beginning some time ago.

params = (nu, T)

Parameters

- **params** – Tupe of length two, specifying (nu, t).
  - nu: the final population size.
  - T: the time in the past at which growth began (in units of 2*Ne generations).
- **ns** – Number of samples in resulting Spectrum. Must be a list of length one.
- **pop_ids** – Optional list of length one specifying the population ID.

moments.Demographics1D.**snm**(*ns*, *pop_ids=None*)

Standard neutral model with theta=1.

Parameters

- **ns** – Number of samples in resulting Spectrum. Must be a list of length one.
- **pop_ids** – Optional list of length one specifying the population ID.

moments.Demographics1D.**three_epoch**(*params*, *ns*, *pop_ids=None*)

Three epoch model of constant sizes.

params = (nuB, nuF, TB, TF)

Parameters

- **params** – Tuple of length four specifying (nuB, nuF, TB, TF).
  - nuB: Ratio of bottleneck population size to ancient pop size.
  - nuF: Ratio of contemporary to ancient pop size.
  - TB: Length of bottleneck (in units of 2*Na generations).
  - TF: Time since bottleneck recovery (in units of 2*Na generations).
- **ns** – Number of samples in resulting Spectrum.
- **pop_ids** – Optional list of length one specifying the population ID.

moments.Demographics1D.**two_epoch**(*params*, *ns*, *pop_ids=None*)

Instantaneous size change some time ago.

params = (nu, T)

Parameters

- **params** – Tuple of length two, specifying (nu, T).
  - nu: the ratio of contemporary to ancient population size.
  - T: the time in the past at which size change happened (in units of 2*Ne generations).
- **ns** – Number of samples in resulting Spectrum. Must be a list of length one.
- **pop_ids** – Optional list of length one specifying the population ID.

Two-population demographic models.

moments.Demographics2D.**IM**(*params*, *ns*, *pop_ids=None*)

Isolation-with-migration model with exponential pop growth.

params = (s, nu1, nu2, T, m12, m21)

ns = [n1, n2]

Parameters

- **params** – Tuple of length 6.
  - s: Size of pop 1 after split. (Pop 2 has size 1-s.)
  - nu1: Final size of pop 1.
  - nu2: Final size of pop 2.
  - T: Time in the past of split (in units of 2*Na generations)
  - m12: Migration from pop 2 to pop 1 (2 * Na * m12)
  - m21: Migration from pop 1 to pop 2
- **ns** – List of population sizes in first and second populations.
- **pop_ids** – List of population IDs.

moments.Demographics2D.**IM_pre**(*params*, *ns*, *pop_ids=None*)

params = (nuPre, TPre, s, nu1, nu2, T, m12, m21)

ns = [n1, n2]

Isolation-with-migration model with exponential pop growth and a size change prior to split.

- nuPre: Size after first size change
- TPre: Time before split of first size change.
- s: Fraction of nuPre that goes to pop1. (Pop 2 has size nuPre*(1-s).)
- nu1: Final size of pop 1.
- nu2: Final size of pop 2.
- T: Time in the past of split (in units of 2*Na generations)
- m12: Migration from pop 2 to pop 1 (2*Na*m12)
- m21: Migration from pop 1 to pop 2
- n1, n2: Sample sizes of resulting Spectrum.

Parameters

- **ns** – List of population sizes in first and second populations.
- **pop_ids** – List of population IDs.

moments.Demographics2D.**bottlegrowth**(*params*, *ns*, *pop_ids=None*)

params = (nuB, nuF, T)

ns = [n1, n2]

Instantanous size change followed by exponential growth with no population split.

- nuB: Ratio of population size after instantanous change to ancient population size
- nuF: Ratio of contempoary to ancient population size
- T: Time in the past at which instantaneous change happened and growth began (in units of 2*Na generations)
- n1, n2: Sample sizes of resulting Spectrum.

Parameters

- **params** – List of parameters, (nuB, nuF, T).
- **ns** – List of population sizes in first and second populations.
- **pop_ids** – List of population IDs.

moments.Demographics2D.**bottlegrowth_split**(*params*, *ns*, *pop_ids=None*)

params = (nuB, nuF, T, Ts)

ns = [n1, n2]

Instantanous size change followed by exponential growth then split.

- nuB: Ratio of population size after instantanous change to ancient population size
- nuF: Ratio of contempoary to ancient population size
- T: Time in the past at which instantaneous change happened and growth began (in units of 2*Na generations)

- Ts: Time in the past at which the two populations split.
- n1, n2: Sample sizes of resulting Spectrum.

Parameters

- **ns** – List of population sizes in first and second populations.
- **pop_ids** – List of population IDs.

moments.Demographics2D.**bottlegrowth_split_mig**(*params*, *ns*, *pop_ids=None*)

params = (nuB, nuF, m, T, Ts) ns = [n1, n2]

Instantanous size change followed by exponential growth then split with migration.

- nuB: Ratio of population size after instantanous change to ancient population size
- nuF: Ratio of contempoary to ancient population size
- m: Migration rate between the two populations (2*Na*m).
- T: Time in the past at which instantaneous change happened and growth began (in units of 2*Na generations)
- Ts: Time in the past at which the two populations split.
- n1, n2: Sample sizes of resulting Spectrum.

Parameters

- **ns** – List of population sizes in first and second populations.
- **pop_ids** – List of population IDs.

moments.Demographics2D.**snm**(*ns*, *pop_ids=None*)

ns = [n1, n2]

Standard neutral model with a split but no divergence.

Parameters

- **ns** – List of population sizes in first and second populations.
- **pop_ids** – List of population IDs.

moments.Demographics2D.**split_mig**(*params*, *ns*, *pop_ids=None*)

Split into two populations of specifed size, with migration.

params = (nu1, nu2, T, m)

ns = [n1, n2]

Parameters

- **params** – Tuple of length 4.
  - nu1: Size of population 1 after split.
  - nu2: Size of population 2 after split.
  - T: Time in the past of split (in units of 2*Na generations)
  - m: Migration rate between populations (2*Na*m)
- **ns** – List of length two specifying sample sizes n1 and n2.
- **pop_ids** – List of population IDs.

Three-population demographic models.

moments.Demographics3D.**out_of_Africa**(*params*, *ns*, *pop_ids=['YRI', 'CEU', 'CHB']*)

The Gutenkunst et al (2009) out-of-Africa that has been reinferred a number of times.

Parameters

- **params** (*list of floats*) – List of parameters, in the order (nuA, TA, nuB, TB, nuEu0, nuEuF, nuAs0, nuAsF, TF, mAfB, mAfEu, mAfAs, mEuAs).
- **ns** (*list of ints*) – List of population sizes in each population, in order given by *pop_ids*.
- **pop_ids** (*list of strings, optional*) – List of population IDs, defaults to ["YRI", "CEU", "CHB"].

15.4 Inference functions

moments.Inference.**ll**(*model*, *data*)

The log-likelihood of the data given the model sfs.

Evaluate the log-likelihood of the data given the model. This is based on Poisson statistics, where the probability of observing k entries in a cell given that the mean number is given by the model is $P(k) = exp(-model) * model^k/k!$.

Note: If either the model or the data is a masked array, the return ll will ignore any elements that are masked in *either* the model or the data.

Parameters

- **model** – The model Spectrum object.
- **data** – The data Spectrum object, with same size as model.

moments.Inference.**ll_multinom**(*model*, *data*)

Log-likelihood of the data given the model, with optimal rescaling.

Evaluate the log-likelihood of the data given the model. This is based on Poisson statistics, where the probability of observing k entries in a cell given that the mean number is given by the model is $P(k) = exp(-model) * model^k/k!$.

model is optimally scaled to maximize ll before calculation.

Note: If either the model or the data is a masked array, the return ll will ignore any elements that are masked in *either* the model or the data.

Parameters

- **model** – The model Spectrum object.
- **data** – The data Spectrum object, with same size as model.

moments.Inference.**optimal_sfs_scaling**(*model*, *data*)

Optimal multiplicative scaling factor between model and data.

This scaling is based on only those entries that are masked in neither model nor data.

Parameters

- **model** – The model Spectrum object.
- **data** – The data Spectrum object, with same size as model.

moments.Inference.**optimally_scaled_sfs**(*model*, *data*)

Optimially scale model sfs to data sfs.

Returns a new scaled model sfs.

Parameters

- **model** – The model Spectrum object.
- **data** – The data Spectrum object, with same size as model.

moments.Inference.**linear_Poisson_residual**(*model*, *data*, *mask=None*)

Return the Poisson residuals, (model - data)/sqrt(model), of model and data.

mask sets the level in model below which the returned residual array is masked. The default of 0 excludes values where the residuals are not defined.

In the limit that the mean of the Poisson distribution is large, these residuals are normally distributed. (If the mean is small, the Anscombe residuals are better.)

Parameters

- **model** – The model Spectrum object.
- **data** – The data Spectrum object, with same size as model.
- **mask** – Optional mask, with same size as model.

moments.Inference.**Anscombe_Poisson_residual**(*model*, *data*, *mask=None*)

Return the Anscombe Poisson residuals between model and data.

mask sets the level in model below which the returned residual array is masked. This excludes very small values where the residuals are not normal. 1e-2 seems to be a good default for the NIEHS human data. (model = 1e-2, data = 0, yields a residual of ~1.5.)

Residuals defined in this manner are more normally distributed than the linear residuals when the mean is small. See this reference below for justification: Pierce DA and Schafer DW, "Residuals in generalized linear models" Journal of the American Statistical Association, 81(396)977-986 (1986).

Note that I tried implementing the "adjusted deviance" residuals, but they always looked very biased for the cases where the data was 0.

Parameters

- **model** – The model Spectrum object.
- **data** – The data Spectrum object, with same size as model.
- **mask** – Optional mask, with same size as model.

moments.Inference.**optimize_log**(*p0*, *data*, *model_func*, *lower_bound=None*, *upper_bound=None*, *verbose=0*, *flush_delay=0.5*, *epsilon=0.001*, *gtol=1e-05*, *multinom=True*, *maxiter=None*, *full_output=False*, *func_args=[]*, *func_kwargs={}*, *fixed_params=None*, *ll_scale=1*, *output_file=None*)

Optimize log(params) to fit model to data using the BFGS method. This optimization method works well when we start reasonably close to the optimum.

Because this works in log(params), it cannot explore values of params < 0. However, it should perform well when parameters range over different orders of magnitude.

Parameters

- **p0** – Initial parameters.
- **data** – Data SFS.

- **model_func** – Function to evaluate model spectrum. Should take arguments `model_func(params, (n1,n2...))`.
- **lower_bound** – Lower bound on parameter values. If not None, must be of same length as p0.
- **upper_bound** – Upper bound on parameter values. If not None, must be of same length as p0.
- **verbose** – If > 0, print optimization status every `verbose` steps.
- **output_file** – Stream verbose output into this filename. If None, stream to standard out.
- **flush_delay** – Standard output will be flushed once every <flush_delay> minutes. This is useful to avoid overloading I/O on clusters.
- **epsilon** – Step-size to use for finite-difference derivatives.
- **gtol** – Convergence criterion for optimization. For more info, see help(scipy.optimize.fmin_bfgs)
- **multinom** – If True, do a multinomial fit where model is optimially scaled to data at each step. If False, assume theta is a parameter and do no scaling.
- **maxiter** – Maximum iterations to run for.
- **full_output** – If True, return full outputs as in described in help(scipy.optimize.fmin_bfgs)
- **func_args** – Additional arguments to model_func. It is assumed that model_func's first argument is an array of parameters to optimize, that its second argument is an array of sample sizes for the sfs, and that its last argument is the list of grid points to use in evaluation. Using func_args. For example, you could define your model function as `def func((p1,p2), ns, f1, f2): ...`. If you wanted to fix f1=0.1 and f2=0.2 in the optimization, you would pass func_args = [0.1,0.2] (and ignore the fixed_params argument).
- **func_kwargs** – Additional keyword arguments to model_func.
- **fixed_params** – If not None, should be a list used to fix model parameters at particular values. For example, if the model parameters are (nu1,nu2,T,m), then fixed_params = [0.5,None,None,2] ll hold nu1=0.5 and m=2. The optimizer will only change T and m. Note that the bounds lists must include all parameters. Optimization will fail if the fixed values lie outside their bounds. A full-length p0 should be passed in; values corresponding to fixed parameters are ignored. For example, suppose your model function is `def func((p1,f1, p2,f2), ns): ...` If you wanted to fix f1=0.1 and f2=0.2 in the optimization, you would pass fixed_params = [None,0.1,None,0.2] (and ignore the func_args argument).
- **ll_scale** – The bfgs algorithm may fail if your initial log-likelihood is too large. (This appears to be a flaw in the scipy implementation.) To overcome this, pass ll_scale > 1, which will simply reduce the magnitude of the log-likelihood. Once in a region of reasonable likelihood, you'll probably want to re-optimize with ll_scale=1.

`moments.Inference.`**optimize_log_fmin**(*p0*, *data*, *model_func*, *lower_bound=None*, *upper_bound=None*, *verbose=0*, *flush_delay=0.5*, *multinom=True*, *maxiter=None*, *maxfun=None*, *full_output=False*, *func_args=[]*, *func_kwargs={}*, *fixed_params=None*, *output_file=None*)

Optimize log(params) to fit model to data using Nelder-Mead. This optimization method may work better than BFGS when far from a minimum. It is much slower, but more robust, because it doesn't use gradient information.

Because this works in log(params), it cannot explore values of params < 0. It should also perform better when parameters range over large scales.

Parameters

- **p0** – Initial parameters.
- **data** – Spectrum with data.
- **model_function** – Function to evaluate model spectrum. Should take arguments (params, (n1,n2...))
- **lower_bound** – Lower bound on parameter values. If not None, must be of same length as p0. A parameter can be declared unbound by assigning a bound of None.
- **upper_bound** – Upper bound on parameter values. If not None, must be of same length as p0. A parameter can be declared unbound by assigning a bound of None.
- **verbose** – If True, print optimization status every <verbose> steps.
- **output_file** – Stream verbose output into this filename. If None, stream to standard out.
- **flush_delay** – Standard output will be flushed once every <flush_delay> minutes. This is useful to avoid overloading I/O on clusters.
- **multinom** – If True, do a multinomial fit where model is optimially scaled to data at each step. If False, assume theta is a parameter and do no scaling.
- **maxiter** – Maximum number of iterations to run optimization.
- **maxfun** – Maximum number of objective function calls to perform.
- **full_output** – If True, return full outputs as in described in help(scipy.optimize.fmin_bfgs)
- **func_args** – Additional arguments to model_func. It is assumed that model_func's first argument is an array of parameters to optimize, that its second argument is an array of sample sizes for the sfs, and that its last argument is the list of grid points to use in evaluation.
- **func_kwargs** – Additional keyword arguments to model_func.
- **fixed_params** – If not None, should be a list used to fix model parameters at particular values. For example, if the model parameters are (nu1,nu2,T,m), then fixed_params = [0.5,None,None,2] will hold nu1=0.5 and m=2. The optimizer will only change T and m. Note that the bounds lists must include all parameters. Optimization will fail if the fixed values lie outside their bounds. A full-length p0 should be passed in; values corresponding to fixed parameters are ignored.

moments.Inference.**optimize_log_powell**(*p0*, *data*, *model_func*, *lower_bound=None*, *upper_bound=None*, *verbose=0*, *flush_delay=0.5*, *multinom=True*, *maxiter=None*, *full_output=False*, *func_args=[]*, *func_kwargs={}*, *fixed_params=None*, *output_file=None*)

Optimize log(params) to fit model to data using Powell's conjugate direction method.

This method works without calculating any derivatives, and optimizes along one direction at a time. May be useful as an initial search for an approximate solution, followed by further optimization using a gradient optimizer.

Because this works in log(params), it cannot explore values of params < 0.

Parameters

- **p0** – Initial parameters.
- **data** – Spectrum with data.
- **model_function** – Function to evaluate model spectrum. Should take arguments (params, (n1,n2...))
- **lower_bound** – Lower bound on parameter values. If not None, must be of same length as p0. A parameter can be declared unbound by assigning a bound of None.

- **upper_bound** – Upper bound on parameter values. If not None, must be of same length as p0. A parameter can be declared unbound by assigning a bound of None.
- **verbose** – If True, print optimization status every <verbose> steps. output_file: Stream verbose output into this filename. If None, stream to standard out.
- **flush_delay** – Standard output will be flushed once every <flush_delay> minutes. This is useful to avoid overloading I/O on clusters. multinom: If True, do a multinomial fit where model is optimially scaled to data at each step. If False, assume theta is a parameter and do no scaling.
- **maxiter** – Maximum iterations to run for.
- **full_output** – If True, return full outputs as in described in help(scipy.optimize.fmin_bfgs)
- **func_args** – Additional arguments to model_func. It is assumed that model_func's first argument is an array of parameters to optimize, that its second argument is an array of sample sizes for the sfs, and that its last argument is the list of grid points to use in evaluation.
- **func_kwargs** – Additional keyword arguments to model_func.
- **fixed_params** – If not None, should be a list used to fix model parameters at particular values. For example, if the model parameters are (nu1,nu2,T,m), then fixed_params = [0.5,None,None,2] will hold nu1=0.5 and m=2. The optimizer will only change T and m. Note that the bounds lists must include all parameters. Optimization will fail if the fixed values lie outside their bounds. A full-length p0 should be passed in; values corresponding to fixed parameters are ignored. (See help(moments.Inference.optimize_log for examples of func_args and fixed_params usage.)

moments.Inference.**optimize_log_lbfgsb**(*p0*, *data*, *model_func*, *lower_bound=None*, *upper_bound=None*, *verbose=0*, *flush_delay=0.5*, *epsilon=0.001*, *pgtol=1e-05*, *multinom=True*, *maxiter=100000.0*, *full_output=False*, *func_args=[]*, *func_kwargs={}*, *fixed_params=None*, *ll_scale=1*, *output_file=None*)

Optimize log(params) to fit model to data using the L-BFGS-B method.

This optimization method works well when we start reasonably close to the optimum. It is best at burrowing down a single minimum. This method is better than optimize_log if the optimum lies at one or more of the parameter bounds. However, if your optimum is not on the bounds, this method may be much slower.

Because this works in log(params), it cannot explore values of params < 0. It should also perform better when parameters range over scales.

The L-BFGS-B method was developed by Ciyou Zhu, Richard Byrd, and Jorge Nocedal. The algorithm is described in:

- R. H. Byrd, P. Lu and J. Nocedal. A Limited Memory Algorithm for Bound Constrained Optimization, (1995), SIAM Journal on Scientific and Statistical Computing , 16, 5, pp. 1190-1208.
- C. Zhu, R. H. Byrd and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization (1997), ACM Transactions on Mathematical Software, Vol 23, Num. 4, pp. 550-560.

Parameters

- **p0** – Initial parameters.
- **data** – Spectrum with data.
- **model_function** – Function to evaluate model spectrum. Should take arguments (params, (n1,n2...))

- **lower_bound** – Lower bound on parameter values. If not None, must be of same length as p0. A parameter can be declared unbound by assigning a bound of None.
- **upper_bound** – Upper bound on parameter values. If not None, must be of same length as p0. A parameter can be declared unbound by assigning a bound of None.
- **verbose** – If > 0, print optimization status every <verbose> steps.
- **output_file** – Stream verbose output into this filename. If None, stream to standard out.
- **flush_delay** – Standard output will be flushed once every <flush_delay> minutes. This is useful to avoid overloading I/O on clusters.
- **epsilon** – Step-size to use for finite-difference derivatives.
- **pgtol** – Convergence criterion for optimization. For more info, see help(scipy.optimize.fmin_l_bfgs_b)
- **multinom** – If True, do a multinomial fit where model is optimially scaled to data at each step. If False, assume theta is a parameter and do no scaling.
- **maxiter** – Maximum algorithm iterations to run.
- **full_output** – If True, return full outputs as in described in help(scipy.optimize.fmin_bfgs)
- **func_args** – Additional arguments to model_func. It is assumed that model_func's first argument is an array of parameters to optimize, that its second argument is an array of sample sizes for the sfs, and that its last argument is the list of grid points to use in evaluation.
- **func_kwargs** – Additional keyword arguments to model_func.
- **fixed_params** – If not None, should be a list used to fix model parameters at particular values. For example, if the model parameters are (nu1,nu2,T,m), then fixed_params = [0.5,None,None,2] will hold nu1=0.5 and m=2. The optimizer will only change T and m. Note that the bounds lists must include all parameters. Optimization will fail if the fixed values lie outside their bounds. A full-length p0 should be passed in; values corresponding to fixed parameters are ignored.
- **ll_scale** – The bfgs algorithm may fail if your initial log-likelihood is too large. (This appears to be a flaw in the scipy implementation.) To overcome this, pass ll_scale > 1, which will simply reduce the magnitude of the log-likelihood. Once in a region of reasonable likelihood, you'll probably want to re-optimize with ll_scale=1.

15.5 Uncertainty functions

Parameter uncertainties and likelihood ratio tests using Godambe information.

moments.Godambe.**FIM_uncert**(*func_ex*, *p0*, *data*, *log=False*, *multinom=True*, *eps=0.01*)

Parameter uncertainties from Fisher Information Matrix. Returns standard deviations of parameter values.

Parameters

- **func_ex** (*demographic model*) – Model function
- **p0** (*list-like*) – Best-fit parameters for func_ex
- **data** (*spectrum object*) – Original data frequency spectrum
- **log** (*bool*) – If True, assume log-normal distribution of parameters. Returned values are then the standard deviations of the *logs* of the parameter values, which can be interpreted as relative parameter uncertainties.

- **multinom** (*bool*) – If True, assume model is defined without an explicit parameter for theta. Because uncertainty in theta must be accounted for to get correct uncertainties for other parameters, this function will automatically consider theta if multinom=True. In that case, the final entry of the returned uncertainties will correspond to theta.
- **eps** (*float*) – Fractional stepsize to use when taking finite-difference derivatives

moments.Godambe.**GIM_uncert**(*func_ex*, *all_boot*, *p0*, *data*, *log=False*, *multinom=True*, *eps=0.01*, *return_GIM=False*)

Parameter uncertainties from Godambe Information Matrix (GIM). Returns standard deviations of parameter values. Bootstrap data is typically generated by splitting the genome into *N* chunks and sampling with replacement from those chunks *N* times.

Parameters

- **func_ex** (*demographic model*) – Model function
- **all_boot** (*list of spectra*) – List of bootstrap frequency spectra
- **p0** (*list-like*) – Best-fit parameters for func_ex
- **data** (*spectrum object*) – Original data frequency spectrum
- **log** (*bool*) – If True, assume log-normal distribution of parameters. Returned values are then the standard deviations of the *logs* of the parameter values, which can be interpreted as relative parameter uncertainties.
- **multinom** (*bool*) – If True, assume model is defined without an explicit parameter for theta. Because uncertainty in theta must be accounted for to get correct uncertainties for other parameters, this function will automatically consider theta if multinom=True. In that case, the final entry of the returned uncertainties will correspond to theta.
- **eps** (*float*) – Fractional stepsize to use when taking finite-difference derivatives
- **return_GIM** – If True, also return the full GIM.

moments.Godambe.**LRT_adjust**(*func_ex*, *all_boot*, *p0*, *data*, *nested_indices*, *multinom=True*, *eps=0.01*)

First-order moment matching adjustment factor for likelihood ratio test.

Parameters

- **func_ex** (*demographic model*) – Model function for complex model
- **all_boot** (*list of spectra*) – List of bootstrap frequency spectra
- **p0** (*list-like*) – Best-fit parameters for the simple model, with nested parameter explicity defined. Although equal to values for simple model, should be in a list form that can be taken in by the complex model you'd like to evaluate.
- **data** (*spectrum object*) – Original data frequency spectrum
- **nested_indices** (*list of ints*) – List of positions of nested parameters in complex model parameter list
- **multinom** (*bool*) – If True, assume model is defined without an explicit parameter for theta. Because uncertainty in theta must be accounted for to get correct uncertainties for other parameters, this function will automatically consider theta if multinom=True.
- **eps** (*float*) – Fractional stepsize to use when taking finite-difference derivatives

15.6 Plotting features

15.6.1 Single-population plotting

moments.Plotting.**plot_1d_comp_Poisson**(*model*, *data*, *fig_num=None*, *residual='Anscombe'*, *plot_masked=False*, *out=None*, *show=True*, *labels=['Model', 'Data']*)

Poisson comparison between 1d model and data.

Parameters

- **model** – 1-dimensional model SFS
- **data** – 1-dimensional data SFS
- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **plot_masked** – Additionally plots (in open circles) results for points in the model or data that were masked.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.
- **labels** – A list of strings of length two, labels for the first and second input frequency spectra. Defaults to "Model" and "Data".

moments.Plotting.**plot_1d_comp_multinom**(*model*, *data*, *fig_num=None*, *residual='Anscombe'*, *plot_masked=False*, *out=None*, *show=True*, *labels=['Model', 'Data']*)

Multinomial comparison between 1d model and data.

Parameters

- **model** – 1-dimensional model SFS
- **data** – 1-dimensional data SFS
- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **plot_masked** – Additionally plots (in open circles) results for points in the model or data that were masked.
- **out** – Output filename to save figure, if given.
- **show** – If True, displays figure. Set to False to supress.

moments.Plotting.**plot_1d_fs**(*fs*, *fig_num=None*, *show=True*, *ax=None*, *out=None*, *ms=3*, *lw=1*)

Plot a 1-dimensional frequency spectrum.

Note that all the plotting is done with pylab. To see additional pylab methods: "import pylab; help(pylab)". Pylab's many functions are documented at http://matplotlib.sourceforge.net/contents.html

Parameters

- **fs** – A single-population Spectrum
- **fig_num** – If used, clear and use figure fig_num for display. If None, a new figure window is created.
- **show** – If True, execute pylab.show command to make sure plot displays.
- **ax** – If None, uses new or specified figure. Otherwise plots in axes object that is given after clearing.
- **out** – If file name is given, saves before showing.

15.6.2 Multi-population plotting

moments.Plotting.**plot_2d_comp_Poisson**(*model*, *data*, *vmin=None*, *vmax=None*, *resid_range=None*, *fig_num=None*, *pop_ids=None*, *residual='Anscombe'*, *adjust=True*, *out=None*, *show=True*)

Poisson comparison between 2d model and data.

Parameters

- **model** – 2-dimensional model SFS
- **data** – 2-dimensional data SFS
- **vmin** – Minimum value plotted.
- **vmax** – Maximum value plotted.
- **resid_range** – Residual plot saturates at +- resid_range.
- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **adjust** – Should method use automatic 'subplots_adjust'? For advanced manipulation of plots, it may be useful to make this False.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_2d_comp_multinom**(*model*, *data*, *vmin=None*, *vmax=None*, *resid_range=None*, *fig_num=None*, *pop_ids=None*, *residual='Anscombe'*, *adjust=True*, *out=None*, *show=True*)

Multinomial comparison between 2d model and data.

Parameters

- **model** – 2-dimensional model SFS
- **data** – 2-dimensional data SFS
- **vmin** – Minimum value plotted.
- **vmax** – Maximum value plotted.
- **resid_range** – Residual plot saturates at +- resid_range.

- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **adjust** – Should method use automatic 'subplots_adjust'? For advanced manipulation of plots, it may be useful to make this False.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_2d_resid**(*resid*, *resid_range=None*, *ax=None*, *pop_ids=None*, *extend='neither'*, *colorbar=True*, *out=None*, *show=True*)

Linear heatmap of 2d residual array.

Parameters

- **sfs** – Residual array to plot.
- **resid_range** – Values > resid range or < resid_range saturate the color spectrum.
- **ax** – Axes object to plot into. If None, the result of pylab.gca() is used.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **extend** – Whether the colorbar should have 'extension' arrows. See help(pylab.colorbar) for more details.
- **colorbar** – Should we plot a colorbar?
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_3d_comp_Poisson**(*model*, *data*, *vmin=None*, *vmax=None*, *resid_range=None*, *fig_num=None*, *pop_ids=None*, *residual='Anscombe'*, *adjust=True*, *out=None*, *show=True*)

Poisson comparison between 3d model and data.

Parameters

- **model** – 3-dimensional model SFS
- **data** – 3-dimensional data SFS
- **vmin** – Minimum value plotted.
- **vmax** – Maximum value plotted.
- **resid_range** – Residual plot saturates at +- resid_range.
- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **adjust** – Should method use automatic 'subplots_adjust'? For advanced manipulation of plots, it may be useful to make this False.
- **out** – Output filename to save figure, if given.

- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_3d_comp_multinom**(*model*, *data*, *vmin=None*, *vmax=None*, *resid_range=None*, *fig_num=None*, *pop_ids=None*, *residual='Anscombe'*, *adjust=True*, *out=None*, *show=True*)

Multinomial comparison between 3d model and data.

Parameters

- **model** – 3-dimensional model SFS
- **data** – 3-dimensional data SFS
- **vmin** – Minimum value plotted.
- **vmax** – Maximum value plotted.
- **resid_range** – Residual plot saturates at +- resid_range.
- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **adjust** – Should method use automatic 'subplots_adjust'? For advanced manipulation of plots, it may be useful to make this False.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_3d_spectrum**(*fs*, *fignum=None*, *vmin=None*, *vmax=None*, *pop_ids=None*, *out=None*, *show=True*)

Logarithmic heatmap of single 3d FS.

Note that this method is slow, because it relies on matplotlib's software rendering. For faster and better looking plots, use plot_3d_spectrum_mayavi.

Parameters

- **fs** – FS to plot
- **vmin** – Values in fs below vmin are masked in plot.
- **vmax** – Values in fs above vmax saturate the color spectrum.
- **fignum** – Figure number to plot into. If None, a new figure will be created.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_4d_comp_Poisson**(*model*, *data*, *vmin=None*, *vmax=None*, *resid_range=None*, *fig_num=None*, *pop_ids=None*, *residual='Anscombe'*, *adjust=True*, *out=None*, *show=True*)

Poisson comparison between 4d model and data.

Parameters

- **model** – 4-dimensional model SFS

- **data** – 4-dimensional data SFS
- **vmin** – Minimum value plotted.
- **vmax** – Maximum value plotted.
- **resid_range** – Residual plot saturates at +- resid_range.
- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **adjust** – Should method use automatic 'subplots_adjust'? For advanced manipulation of plots, it may be useful to make this False.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_4d_comp_multinom**(*model*, *data*, *vmin=None*, *vmax=None*, *resid_range=None*, *fig_num=None*, *pop_ids=None*, *residual='Anscombe'*, *adjust=True*, *out=None*, *show=True*)

Multinomial comparison between 4d model and data.

Parameters

- **model** – 4-dimensional model SFS
- **data** – 4-dimensional data SFS
- **vmin** – Minimum value plotted.
- **vmax** – Maximum value plotted.
- **resid_range** – Residual plot saturates at +- resid_range.
- **fig_num** – Clear and use figure fig_num for display. If None, an new figure window is created.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **residual** – 'Anscombe' for Anscombe residuals, which are more normally distributed for Poisson sampling. 'linear' for the linear residuals, which can be less biased.
- **adjust** – Should method use automatic 'subplots_adjust'? For advanced manipulation of plots, it may be useful to make this False.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

moments.Plotting.**plot_single_2d_sfs**(*sfs*, *vmin=None*, *vmax=None*, *ax=None*, *pop_ids=None*, *extend='neither'*, *colorbar=True*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *out=None*, *show=True*)

Heatmap of single 2d SFS.

If vmax is greater than a factor of 10, plot on log scale.

Returns colorbar that is created.

Parameters

- **sfs** – SFS to plot
- **vmin** – Values in sfs below vmin are masked in plot.
- **vmax** – Values in sfs above vmax saturate the color spectrum.
- **ax** – Axes object to plot into. If None, the result of pylab.gca() is used.
- **pop_ids** – If not None, override pop_ids stored in Spectrum.
- **extend** – Whether the colorbar should have 'extension' arrows. See help(pylab.colorbar) for more details.
- **colorbar** – Should we plot a colorbar?
- **cmap** – Pylab colormap to use for plotting.
- **out** – Output filename to save figure, if given.
- **show** – If True, execute pylab.show command to make sure plot displays.

CHAPTER

SIXTEEN

API FOR LINKAGE DISEQUILIBRIUM

16.1 LD statistics class and function

class moments.LD.**LDstats**(*data*, *num_pops=None*, *pop_ids=None*)

Represents linkage disequilibrium statistics as a list of arrays, where each entry in the list is an array of statistics for a corresponding recombination rate. The final entry in the list is always the heterozygosity statistics. Thus, if we have an LDstats object for 3 recombination rate values, the list will have length 4.

LDstats are represented as a list of statistics over two locus pairs for a given recombination distance.

Parameters

- **data** (*list of arrays*) – A list of LD and heterozygosity stats.
- **num_pops** (*int*) – Number of populations. For one population, higher order statistics may be computed.
- **pop_ids** (*list of strings, optional*) – Population IDs in order that statistics are represented here.

H(*pops=None*)

Returns heterozygosity statistics for the populations given.

Parameters

pops (*list of ints, optional*) – The indexes of populations to return stats for.

LD(*pops=None*)

Returns LD stats for populations given (if None, returns all).

Parameters

pops (*list of ints, optional*) – The indexes of populations to return stats for.

admix(*pop0*, *pop1*, *f*, *new_id='Adm'*)

Admixture between pop0 and pop1, given by indexes. f is the fraction contributed by pop0, so pop1 contributes 1-f. If new_id is not specified, the admixed population's name is 'Adm'. Otherwise, we can set it with new_id=new_pop_id.

Parameters

- **pop0** (*int*) – First population to admix.
- **pop1** (*int*) – Second population to admix.
- **f** (*float*) – The fraction of ancestry contributed by pop0, so pop1 contributes 1 - f.
- **new_id** (*str, optional*) – The name of the admixed population.

f2(*X*, *Y*)

Returns $f_2(X, Y) = (X - Y)^2$.

X, and Y can be specified as population ID strings, or as indexes (but these cannot be mixed).

Parameters

- **X** – One of the populations, as index or population ID.
- **Y** – The other population, as index or population ID.

f3(*X*, *Y*, *Z*)

Returns $f_3(X; Y, Z) = (X - Y)(X - Z)$. A significantly negative f_3 of this form suggests that population X is the result of admixture between ancient populations related to Y and Z. A positive value suggests that X is an outgroup to Y and Z.

X, Y, and Z can be specified as population ID strings, or as indexes (but these cannot be mixed).

Parameters

- **X** – The "test" population, as index or population ID.
- **Y** – The first reference population, as index or population ID.
- **Z** – The second reference population, as index or population ID.

f4(*X*, *Y*, *Z*, *W*)

Returns $f_4(X, Y; Z, W) = (X - Y)(Z - W)$.

X, Y, and Z can be specified as population ID strings, or as indexes (but these cannot be mixed).

Parameters

- **X** – The "test" population, as index or population ID.
- **Y** – The first reference population, as index or population ID.
- **Z** – The second reference population, as index or population ID.
- **W** –

static from_demes(*g*, *sampled_demes*, *sample_times=None*, *rho=None*, *theta=0.001*, *r=None*, *u=None*, *Ne=None*)

Takes a deme graph and computes the LD stats. `demes` is a package for specifying demographic models in a user-friendly, human-readable YAML format. This function automatically parses the demographic description and returns a LD for the specified populations and recombination and mutation rates.

Parameters

- **g** (`demes.DemeGraph`) – A `demes` DemeGraph from which to compute the LD.
- **sampled_demes** (*list of strings*) – A list of deme IDs to take samples from. We can repeat demes, as long as the sampling of repeated deme IDs occurs at distinct times.
- **sample_times** (*list of floats, optional*) – If None, assumes all sampling occurs at the end of the existence of the sampled deme. If there are ancient samples, `sample_times` must be a list of same length as `sampled_demes`, giving the sampling times for each sampled deme. Sampling times are given in time units of the original deme graph, so might not necessarily be generations (e.g. if `g.time_units` is years)
- **rho** – The population-size scaled recombination rate(s). Can be None, a non-negative float, or a list of values. Cannot be used with `Ne`.
- **theta** – The population-size scaled mutation rate. Cannot be used with `Ne`.

- **r** – The raw recombination rate. Can be None, a non-negative float, or a list of values. Must be used with `Ne`.
- **u** – The raw per-base mutation rate. Must be used with `Ne`, in which case `theta` is set to `4 * Ne * u`.
- **Ne** (*float, optional*) – The reference population size. If none is given, we use the initial size of the root deme. For use with `r` and `u`, to compute `rho` and `theta`. If `rho` and/or `theta` are given, we do not pass Ne.

Returns
A `moments.LD` LD statistics object, with number of populations equal to the length of `sampled_demes`.

Return type
moments.LD.LDstats

static from_file(*fid*, *return_statistics=False*, *return_comments=False*)

Read LD statistics from file.

Parameters

- **fid** (*str*) – The file name to read from or an open file object.
- **return_statistics** (*bool, optional*) – If true, returns statistics writen to file.
- **return_comments** (*bool, optional*) – If true, the return value is (y, comments), where comments is a list of strings containing the comments from the file (without #'s).

integrate(*nu*, *tf*, *dt=0.001*, *rho=None*, *theta=0.001*, *m=None*, *selfing=None*, *selfing_rate=None*, *frozen=None*)

Integrates the LD statistics forward in time. When integrating LD statistics for a list of recombination rates and mutation rate, they must be passed as keywork arguments to this function. We can integrate either single-population LD statistics up to order 10, or multi-population LD statistics but only for order 2 (which includes D^2, Dz, and π_2).

Parameters

- **nu** (*list or function*) – The relative population size, may be a function of time, given as a list [nu1, nu2, ...]
- **tf** (*float*) – Total time to integrate
- **dt** (*float*) – Integration timestep
- **rho** (*float or list of floats*) – Can be a single recombination rate or list of recombination rates (in which case we are integrating a list of LD stats for each rate)
- **theta** – The per base population-scaled mutation rate (4N*mu) if we pass [theta1, theta2], differing mutation rates at left and right locus, implemented in the ISM=True model
- **m** (*array*) – The migration matrix (num_pops x num_pops, storing m_ij migration rates where m_ij is probability that a lineage in i had parent in j m_ii is unused, and found by summing off diag elements in the ith row
- **selfing** (*list of floats*) – A list of selfing probabilities, same length as nu.
- **selfing_rate** (*list of floats*) – Alias for selfing.
- **frozen** (*list of bools*) – A list of True and False same length as nu. True implies that a lineage is frozen (as in ancient samples). False integrates as normal.

marginalize(*pops*)

Marginalize over the LDstats, removing moments for given populations.

Parameters
pops (*int or list of ints*) – The index or list of indexes of populations to marginalize.

merge(*pop0*, *pop1*, *f*, *new_id='Merged'*)

Merger of populations pop0 and pop1, with fraction f from pop0 and 1-f from pop1. Places new population at the end, then marginalizes pop0 and pop1. To admix two populations and keep one or both, use pulse migrate or admix, respectively.

Parameters

- **pop0** (*int*) – First population to merge.
- **pop1** (*int*) – Second population to merge.
- **f** (*float*) – The fraction of ancestry contributed by pop0, so pop1 contributes 1 - f.
- **new_id** (*str, optional*) – The name of the merged population.

names()

Returns the set of LD and heterozygosity statistics names for the number of populations represented by the LDstats.

Note that this will always return the full set of statistics,

pulse_migrate(*pop0*, *pop1*, *f*)

Pulse migration/admixure event from pop0 to pop1, with fraction f replacement. We use the admix function above. We want to keep the original population names the same, if they are given in the LDstats object, so we use new_pop=self.pop_ids[pop1].

We admix pop0 and pop1 with fraction f and 1-f, then swap the new admixed population with pop1, then marginalize the original pop1.

Parameters

- **pop0** (*int*) – The index of the source population.
- **pop1** (*int*) – The index of the target population.
- **f** (*float*) – The fraction of ancestry contributed by the source population.

split(*pop_to_split*, *new_ids=None*)

Splits the population given into two child populations. One child population keeps the same index and the second child population is placed at the end of the list of present populations. If new_ids is given, we can set the population IDs of the child populations, but only if the input LDstats have population IDs available.

Parameters

- **pop_to_split** (*int*) – The index of the population to split.
- **new_ids** (*list of strings, optional*) – List of child population names, of length two.

static steady_state(*nus*, *m=None*, *rho=None*, *theta=0.001*, *selfing_rate=None*, *pop_ids=None*)

Computes the steady state solution for one or two populations. The number of populations is determined by the length of `nus`, which is a list with relative population sizes (often, these will be set to 1, meaning sizes are equal to some reference or ancestral population size).

The steady state can only be found for one- and two-population scenarios. If two populations are desired, we must provide `m`, a 2-by-2 migration matrix, and there must be at least one nonzero migration rate. This corresponds to an island model with asymmetric migration and potentially unequal population sizes.

Parameters

- **nus** (*list of numbers*) – The relative population sizes, with one or two entries, corresponding to a steady state solution with one or two populations, resp.
- **m** (*array-like*) – A migration matrix, only provided when the length of *nus* is 2.
- **rho** – The population-size scaled recombination rate(s). Can be None, a non-negative float, or a list of values.
- **theta** (*float*) – The population-size scaled mutation rate
- **selfing_rate** (*number or list of numbers*) – Self-fertilization rate(s), given as a number (for a single population, or list of numbers (for two populations). Selfing rates must be between 0 and 1.
- **pop_ids** (*list of strings*) – The population IDs.

Returns
A `moments.LD` LD statistics object.

Return type
moments.LD.LDstats

swap_pops(*pop0*, *pop1*)

Swaps pop0 and pop1 in the order of the population in the LDstats.

Parameters

- **pop0** (*int*) – The index of the first population to swap.
- **pop1** (*int*) – The index of the second population to swap.

to_file(*fid*, *precision=16*, *statistics='ALL'*, *comment_lines=[]*)

Write LD statistics to file.

The file format is:

- # Any number of comment lines beginning with a '#'
- A single line containing an integer giving the number of populations.
- On the *same line*, optional, the names of those populations. If names are given, there needs to be the same number of pop_ids as the integer number of populations. For example, the line could be '3 YRI CEU CHB'.
- A single line giving the names of the *LD* statistics, in the order they appear for each recombination rate distance or bin. Optionally, this line could read ALL, indicating that every statistic in the basis is given, and in the 'correct' order.
- A single line giving the names of the *heterozygosity* statistics, in the order they appear in the final row of data. Optionally, this line could read ALL.
- A line giving the number of recombination rate bins/distances we have data for (so we know how many to read)
- One line for each row of LD statistics.
- A single line for the heterozygosity statistics.

Parameters

- **fid** (*str*) – The file name to write to or an open file object.

- **precision** (*int*) – The precision with which to write out entries of the LD stats. (They are formated via %.<p>g, where <p> is the precision.)
- **statistics** (*list of list of strings*) – Defaults to 'ALL', meaning all statistics are given in the LDstats object. Otherwise, list of two lists, first giving present LD stats, and the second giving present het stats.
- **comment_lines** (*list of srtings*) – List of strings to be used as comment lines in the header of the output file. I use comment lines mainly to record the recombination bins or distances given in the LDstats (something like "'edges = ' + str(r_edges)".

16.2 Demographic functions

moments.LD.Demographics1D.**bottlegrowth**(*params*, *order=2*, *rho=None*, *theta=0.001*, *pop_ids=None*)

Exponential growth (or decay) model after size change.

Parameters

- **params** (*list*) – The relative initial and final sizes of the final epoch and its integration time in genetic units: (nuB, nuF, T).
- **order** (*int*) – The maximum order of the LD statistics. Defaults to 2.
- **rho** (*float or list of floats, optional*) – Population-scaled recombination rate (4Nr), given as scalar or list of rhos.
- **theta** (*float*) – Population-scaled mutation rate (4Nu). Defaults to 0.001.
- **pop_ids** (*lits of str, optional*) – List of population IDs of length 1.

moments.LD.Demographics1D.**growth**(*params*, *order=2*, *rho=None*, *theta=0.001*, *pop_ids=None*)

Exponential growth (or decay) model.

Parameters

- **params** (*list*) – The relative final size and integration time of recent epoch, in genetic units: (nuF, T)
- **order** (*int*) – The maximum order of the LD statistics. Defaults to 2.
- **rho** (*float or list of floats, optional*) – Population-scaled recombination rate (4Nr), given as scalar or list of rhos.
- **theta** (*float*) – Population-scaled mutation rate (4Nu). Defaults to 0.001.
- **pop_ids** (*lits of str, optional*) – List of population IDs of length 1.

moments.LD.Demographics1D.**snm**(*params=None*, *order=2*, *rho=None*, *theta=0.001*, *pop_ids=None*)

Equilibrium neutral model. Does not take demographic parameters.

Parameters

- **params** – Unused.
- **order** (*int*) – The maximum order of the LD statistics. Defaults to 2.
- **rho** (*float or list of floats, optional*) – Population-scaled recombination rate (4Nr), given as scalar or list of rhos.
- **theta** (*float*) – Population-scaled mutation rate (4Nu). Defaults to 0.001.
- **pop_ids** (*lits of str, optional*) – List of population IDs of length 1.

moments.LD.Demographics1D.**three_epoch**(*params*, *order=2*, *rho=None*, *theta=0.001*, *pop_ids=None*)

Three epoch model with constant sized epochs.

Parameters

- **params** (*list*) – The relative sizes and integration times of recent epochs, in genetic units: (nu1, nu2, T1, T2).
- **order** (*int*) – The maximum order of the LD statistics. Defaults to 2.
- **rho** (*float or list of floats, optional*) – Population-scaled recombination rate (4Nr), given as scalar or list of rhos.
- **theta** (*float*) – Population-scaled mutation rate (4Nu). Defaults to 0.001.
- **pop_ids** (*lits of str, optional*) – List of population IDs of length 1.

moments.LD.Demographics1D.**two_epoch**(*params*, *order=2*, *rho=None*, *theta=0.001*, *pop_ids=None*)

Two epoch model with a single size change and constant sized epochs.

Parameters

- **params** (*list*) – The relative size and integration time of recent epoch, in genetic units: (nu, T).
- **order** (*int*) – The maximum order of the LD statistics. Defaults to 2.
- **rho** (*float or list of floats, optional*) – Population-scaled recombination rate (4Nr), given as scalar or list of rhos.
- **theta** (*float*) – Population-scaled mutation rate (4Nu). Defaults to 0.001.
- **pop_ids** (*lits of str, optional*) – List of population IDs of length 1.

moments.LD.Demographics2D.**snm**(*params=None*, *rho=None*, *theta=0.001*, *pop_ids=None*)

Equilibrium neutral model. Neutral steady state followed by split in the immediate past.

Parameters

- **params** – Unused.
- **rho** (*float or list of floats, optional*) – Population-scaled recombination rate (4Nr), given as scalar or list of rhos.
- **theta** (*float*) – Population-scaled mutation rate (4Nu). Defaults to 0.001.
- **pop_ids** (*lits of str, optional*) – List of population IDs of length 2.

moments.LD.Demographics2D.**split_mig**(*params*, *rho=None*, *theta=0.001*, *pop_ids=None*)

Split into two populations of specifed size, which then have their own relative constant sizes and symmetric migration between populations.

- nu1: Size of population 1 after split.
- nu2: Size of population 2 after split.
- T: Time in the past of split (in units of 2*Na generations)
- m: Migration rate between populations (2*Na*m)

Parameters

- **params** – The input parameters: (nu1, nu2, T, m)
- **rho** (*float or list of floats, optional*) – Population-scaled recombination rate (4Nr), given as scalar or list of rhos.

- **theta** (*float*) – Population-scaled mutation rate (4Nu). Defaults to 0.001.
- **pop_ids** (*lits of str, optional*) – List of population IDs of length 1.

Three-population demographic models.

moments.LD.Demographics3D.**out_of_Africa**(*params*, *rho=None*, *theta=0.001*, *pop_ids=['YRI', 'CEU', 'CHB']*)

The Gutenkunst et al (2009) out-of-Africa that has been reinferred a number of times.

Parameters

- **params** – List of parameters, in the order (nuA, TA, nuB, TB, nuEu0, nuEuF, nuAs0, nuAsF, TF, mAfB, mAfEu, mAfAs, mEuAs).
- **rho** – Recombination rate or list of recombination rates (population-size scaled).
- **theta** – Population-size scaled mutation rate.
- **pop_ids** – List of population IDs.

16.3 Utility functions

moments.LD.Util.**het_names**(*num_pops*)

Returns the heterozygosity statistic representation names.

Parameters

num_pops (*int*) – Number of populations.

moments.LD.Util.**ld_names**(*num_pops*)

Returns the LD statistic representation names.

Parameters

num_pops (*int*) – Number of populations.

moments.LD.Util.**map_moment**(*mom*)

There are repeated moments with equal expectations, so we collapse them into the same moment.

Parameters

mom (*str*) – The moment to map to its "canonical" name.

moments.LD.Util.**moment_names**(*num_pops*)

Returns a tuple of length two with LD and heterozygosity moment names.

Parameters

num_pops (*int*) – Number of populations

moments.LD.Util.**perturb_params**(*params*, *fold=1*, *lower_bound=None*, *upper_bound=None*)

Generate a perturbed set of parameters. Each element of params is randomly perturbed by the given factor of 2 up or down.

Parameters

- **params** (*list*) – A list of input parameters.
- **fold** (*float*) – Number of factors of 2 to perturb by.
- **lower_bound** (*list*) – If not None, the resulting parameter set is adjusted to have all value greater than lower_bound. Must have equal length to `params`.

- **upper_bound** (*list*) – If not None, the resulting parameter set is adjusted to have all value less than upper_bound. Must have equal length to params.

moments.LD.Util.**rescale_params**(*params*, *types*, *Ne=None*, *gens=1*, *uncerts=None*, *time_offset=0*)

Rescale parameters to physical units, so that times are in generations or years, sizes in effective instead of relative sizes, and migration probabilities in per-generation units.

For generation times of events to be correctly rescaled, times in the parameters list must be specified so that earlier epochs are earlier in the list, because we return rescaled cumulative times. All time parameters must refer to consecutive epochs. Epochs need not start at contemporary time, and we can specify the time offset using *time_offset*.

If uncertainties are not given (*uncerts = None*), the return value is an array of rescaled parameters. If uncertainties are given, the return value has length two: the first entry is an array of rescaled parameters, and the second entry is an array of rescaled uncertainties.

Parameters

- **params** (*list*) – List of parameters.
- **types** (*list*) – List of parameter types. Times are given by "T", sizes by "nu", effective size by "Ne", migration rates by "m", and fractions by "x" or "f".
- **Ne** (*float*) – The effective population size, typically as the last entry in params.
- **gens** (*float*) – The generation time.
- **uncerts** (*list*) – List of uncertainties, same length as params.
- **time_offset** (*int or float*) – The amount of time added to each rescaled time point. This lets us have consecutive epochs that stop short of time 0 (final sampling time).

16.4 Parsing functions

moments.LD.Parsing.**bootstrap_data**(*all_data*, *normalization=0*)

Returns bootstrapped variances for LD statistics. This function operates on data that is sums (i.e. the direct output of compute_ld_statistics()), instead of mean statistics.

We first check that all 'stats', 'bins', 'pops' (if present), match across all regions

If there are N total regions, we compute N bootstrap replicates by sampling N times with replacement and summing over all 'sums'.

Parameters

- **all_data** (*dict*) – A dictionary (with arbitrary keys), where each value is LD statistics computed from a distinct region. all_data[reg] stats from each region has keys, 'bins', 'sums', 'stats', and optional 'pops'.
- **normalization** (*int*) – we work with σ_d^2 statistics, and by default we use population 0 to normalize stats

moments.LD.Parsing.**compute_average_stats**(*Gs*, *genotypes=True*)

Takes the outputs of compute_pairwise_stats and returns the average value for each statistic.

Parameters

- **Gs** – A genotype matrix, of size L-by-n, where L is the number of loci and n is the sample size. Missing data is encoded as -1.
- **genotypes** – If True, use 0, 1, 2 genotypes. If False, use 0, 1 phased haplotypes.

moments.LD.Parsing.**compute_average_stats_between**(*Gs1*, *Gs2*, *genotypes=True*)

Takes the outputs of compute_pairwise_stats_between and returns the average value for each statistic.

Parameters

- **Gs1** – A genotype matrices, of size L1 by n, where L1 is the number of loci and n is the sample size. Missing data is encoded as -1.
- **Gs2** – A genotype matrices, of size L2 by n, where L1 is the number of loci and n is the sample size. Missing data is encoded as -1.

moments.LD.Parsing.**compute_ld_statistics**(*vcf_file*, *bed_file=None*, *chromosome=None*, *rec_map_file=None*, *map_name=None*, *map_sep=None*, *pop_file=None*, *pops=None*, *cM=True*, *r_bins=None*, *bp_bins=None*, *min_bp=None*, *use_genotypes=True*, *use_h5=True*, *stats_to_compute=None*, *ac_filter=True*, *report=True*, *report_spacing=1000*, *use_cache=True*)

Computes LD statistics for a given VCF. Binning can be done by base pair or recombination distances, the latter requiring a recombination map. For more than one population, we include a population file that maps samples to populations, and specify with populations to compute statistics fro.

If data is phased, we can set `use_genotypes` to False, and there are other options for masking data.

Note: Currently, the recombination map is not given in HapMap format. Future versions will accept HapMap formatted recombination maps and deprecate some of the boutique handling of map options here.

Parameters

- **vcf_file** (*str*) – The input VCF file name.
- **bed_file** (*str*) – An optional bed file that specifies regions over which to compute LD statistics. If None, computes statistics for all positions in VCF.
- **chromosome** (*str*) – If None, treats all positions in VCF as coming from same chromosome. If multiple chromosomes are reported in the same VCF, we need to specify which chromosome to keep variants from.
- **rec_map_file** (*str*) – The input recombination map. The format is {pos} {map (cM)} {additional maps}
- **map_name** (*str*) – If None, takes the first map column, otherwise takes the specified map column with the name matching the recombination map file header.
- **map_sep** (*str*) – Deprecated! We now read the recombination map, splitting by any white space. Previous behaviour: Tells pandas how to parse the recombination map.
- **pop_file** (*str*) – A file the specifies the population for each sample in the VCF. Each sample is listed on its own line, in the format "{sample} {pop}". The first line must be "sample pop".
- **pops** (*list(str)*) – List of populations to compute statistics for. If none are given, it treates every sample as coming from the same population.
- **cM** (*bool*) – If True, the recombination map is specified in cM. If False, the map is given in units of Morgans.
- **r_bins** (*list(float)*) – A list of raw recombination rate bin edges.
- **bp_bins** (*list(float)*) – If `r_bins` are not given, a list of bp bin edges (for use when no recombination map is specified).

- **min_bp** (*int, float*) – The minimum bp allowed for a segment specified by the bed file.
- **use_genotypes** (*bool*) – If True, we assume the data in the VCF is unphased. Otherwise, we use phased information.
- **use_h5** (*bool*) – If True, we use the h5 format.
- **stats_to_compute** (*list*) – If given, we compute only the statistics specified. Otherwise, we compute all possible statistics for the populations given.
- **ac_filter** – Ensure at least two samples are present per population. This prevents computed heterozygosity statistics from returning NaN when some loci have too few called samples.
- **report** (*bool*) – If True, we report the progress of our parsing.
- **report_spacing** (*int*) – We track the number of "left" variants we compute, and report our progress with the given spacing.
- **use_cache** (*bool*) – If True, cache intermediate results.

moments.LD.Parsing.**compute_pairwise_stats**(*Gs*, *genotypes=True*)

Computes D^2, Dz, π_2, and D for every pair of loci within a block of SNPs, coded as a genotype matrix.

Parameters

- **Gs** – A genotype matrix, of size L-by-n, where L is the number of loci and n is the sample size. Missing data is encoded as -1.
- **genotypes** – If True, use 0, 1, 2 genotypes. If False, use 0, 1 phased haplotypes.

moments.LD.Parsing.**compute_pairwise_stats_between**(*Gs1*, *Gs2*, *genotypes=True*)

Computes D^2, Dz, π_2, and D for every pair of loci between two blocks of SNPs, coded as genotype matrices.

The Gs are matrices, where rows correspond to loci and columns to individuals. Both matrices must have the same number of individuals. If Gs1 has length L1 and Gs2 has length L2, we compute all pairwise counts, which has size (L1*L2, 9).

We use the sparse genotype matrix representation, where we first "sparsify" the genotype matrix, and then count two-locus genotype configurations from that, from which we compute two-locus statistics

Parameters

- **Gs1** – A genotype matrices, of size L1 by n, where L1 is the number of loci and n is the sample size. Missing data is encoded as -1.
- **Gs2** – A genotype matrices, of size L2 by n, where L1 is the number of loci and n is the sample size. Missing data is encoded as -1.
- **genotypes** – If True, use 0, 1, 2 genotypes. If False, use 0, 1 phased haplotypes.

moments.LD.Parsing.**get_bootstrap_sets**(*all_data*, *num_bootstraps=None*, *normalization=0*, *remove_norm_stats=True*)

From a dictionary of all the regional data, resample with replacement to construct bootstrap data.

Returns a list of bootstrapped datasets of mean statistics.

Parameters

- **all_data** (*dict*) – Dictionary of regional LD statistics. Keys are region identifiers and must be unique, and the items are the outputs of `compute_ld_statistics`.
- **num_bootstraps** (*int*) – The number of bootstrap replicates to compute. If None, it computes the same number as the nubmer of regions in `all_data`.
- **normalization** (*int*) – The index of the population to normalize by. Defaults to 0.

moments.LD.Parsing.**get_genotypes**(*vcf_file*, *bed_file=None*, *chromosome=None*, *min_bp=None*, *use_h5=True*, *report=True*)

Given a vcf file, we extract the biallelic SNP genotypes. If bed_file is None, we use all valid variants. Otherwise we filter genotypes by the given bed file. If chromosome is given, filters to keep snps only in given chrom (useful for vcfs spanning multiple chromosomes).

If use_h5 is True, we try to load the h5 file, which has the same path/name as vcf_file, but with {fname}.h5 instead of {fname}.vcf or {fname}.vcf.gz. If the h5 file does not exist, we create it and save it as {fname}.h5

Returns (biallelic positions, biallelic genotypes, biallelic allele counts, sampled ids).

Parameters

- **vcf_file** (*str*) – A VCF-formatted file.
- **bed_file** (*str, optional*) – A bed file specifying regions to compute statistics from. The chromosome name formatting must match the chromosome name formatting of the input VCF (i.e., both carry the leading "chr" or both omit it).
- **min_bp** (*int, optional*) – only used with bed file, filters out features that are smaller than min_bp.
- **chromosome** (*int or str, optional*) – Chromosome to compute LD statistics from.
- **use_h5** (*bool, optional*) – If use_h5 is True, we try to load the h5 file, which has the same path/name as vcf_file, but with .h5 instead of .vcf or .vcf.gz extension. If the h5 file does not exist, we create it and save it with .h5 extension. Defaults to True.
- **report** (*bool, optional*) – Prints progress updates if True, silent otherwise. Defaults to True.

moments.LD.Parsing.**means_from_region_data**(*all_data*, *stats*, *norm_idx=0*)

Get means over all parsed regions.

Parameters

- **all_data** (*dict*) – A dictionary with keys as unique identifiers of the regions and values as reported stats from compute_ld_statistics.
- **stats** (*list of lists*) – The list of LD and H statistics that are present in the data replicates.
- **norm_idx** (*int, optional*) – The index of the population to normalize by.

moments.LD.Parsing.**subset_data**(*data*, *pops_to*, *normalization=0*, *r_min=None*, *r_max=None*, *remove_Dz=False*)

Take the output data and get r_edges, ms, vcs, and stats to pass to inference machinery. pops_to are the subset of the populations to marginalize the data to. r_min and r_max trim bins that fall outside of this range, and remove_Dz allows us to remove all σ_{Dz} statistics.

Parameters

- **data** – The output of bootstrap_data, which contains bins, statistics, populations, means, and variance-covariance matrices.
- **pops_to** – A list of populations to subset to.
- **normalization** – The population index that the original data was normalized by.
- **r_min** – The minimum recombination distance to keep.
- **r_max** – The maximum recombination distance to keep.
- **remove_Dz** – If True, remove all Dz statistics. Otherwise keep them.

16.5 Inference and computing confidence intervals

16.5.1 Inference methods

moments.LD.Inference.**bin_stats**(*model_func*, *params*, *rho=[]*, *theta=0.001*, *spread=None*, *kwargs={}*)

Computes LD statist for a given model function over bins defined by `rho`. Here, `rho` gives the bin edges, and we assume no spaces between bins. That is, if the length of the input recombination rates is l, the number of bins is $l-1$.

Parameters

- **model_func** – The model function that takes parameters in the form `model_func(params, rho=rho, theta=theta, **kwargs)`.
- **params** (*list of floats*) – The parameters to evaluate the model at.
- **rho** (*list of floats*) – The scaled recombination rate bin edges.
- **theta** (*float, optional*) – The mutation rate
- **spread** (*list of arrays*) – A list of length rho-1 (number of bins), where each entry is an array of length rho+1 (number of bins plus amount outside bin range to each side). Each array must sum to one.
- **kwargs** – Extra keyword arguments to pass to `model_func`.

moments.LD.Inference.**ll_over_bins**(*xs*, *mus*, *Sigmas*)

Compute the composite log-likelihood over LD and heterozygosity statistics, given data and expectations. Inputs must be in the same order, and we assume each bin is independent, so we sum _ll(x, mu, Sigma) over each bin.

Parameters

- **xs** – A list of data arrays.
- **mus** – A list of model function output arrays, same length as `xs`.
- **Sigmas** – A list of var-cov matrices, same length as `xs`.

moments.LD.Inference.**optimize_log_fmin**(*p0*, *data*, *model_func*, *rs=None*, *theta=None*, *u=2e-08*, *Ne=None*, *lower_bound=None*, *upper_bound=None*, *verbose=0*, *flush_delay=0.5*, *normalization=0*, *func_args=[]*, *func_kwargs={}*, *fixed_params=None*, *use_afs=False*, *Leff=None*, *multinom=False*, *ns=None*, *statistics=None*, *pass_Ne=False*, *spread=None*, *maxiter=None*, *maxfun=None*)

Optimize (using the log of) the parameters using a downhill simplex algorithm. Initial parameters `p0`, the data `[means, varcovs]`, the demographic `model_func`, and `rs` to specify recombination bin edges are required. `Ne` must either be specified as a keyword argument or is included as the *last* parameter in `p0`.

Parameters

- **p0** (*list*) – The initial guess for demographic parameters, demography parameters plus (optionally) Ne.
- **data** (*list*) – The parsed data[means, varcovs, fs]. The frequency spectrum fs is optional, and used only if use_afs=True.
  - Means: The list of mean statistics within each bin (has length `len(rs)` or `len(rs) - 1` if using AFS). If we are not using the AFS, which is typical, the heterozygosity statistics come last.
  - varcovs: The list of varcov matrices matching the data in `means`.

- **model_func** (*list*) – The demographic model to compute statistics for a given rho. If we are using AFS, it's a list of the two models [LD func, AFS func]. If we're using LD stats alone, we pass a single LD model as a list: [LD func].
- **rs** (*list*) – The list of raw recombination rates defining bin edges.
- **theta** (*float, optional*) – The population scaled per base mutation rate (4*Ne*mu, not 4*Ne*mu*L).
- **u** (*float, optional*) – The raw per base mutation rate. Cannot be used with `theta`.
- **Ne** (*float, optional*) – The fixed effective population size to scale u and r. If `Ne` is a parameter to fit, it should be the last parameter in `p0`.
- **lower_bound** (*list, optional*) – Defaults to `None`. Constraints on the lower bounds during optimization. These are given as lists of the same length of the parameters.
- **upper_bound** (*list, optional*) – Defaults to `None`. Constraints on the upper bounds during optimization. These are given as lists of the same length of the parameters.
- **verbose** (*int, optional*) – If an integer greater than 0, prints updates of the optimization procedure at intervals given by that spacing.
- **func_args** (*list, optional*) – Additional arguments to be passed to `model_func`.
- **func_kwargs** (*dict, optional*) – Additional keyword arguments to be passed to `model_func`.
- **fixed_params** (*list, optional*) – Defaults to `None`. To fix some parameters, this should be a list of equal length as `p0`, with `None` for parameters to be fit and fixed values at corresponding indexes.
- **use_afs** (*bool, optional*) – Defaults to `False`. We can pass a model to compute the frequency spectrum and use that instead of heterozygosity statistics for single-locus data.
- **Leff** (*float, optional*) – The effective length of genome from which the fs was generated (only used if fitting to afs).
- **multinom** (*bool, optional*) – Only used if we are fitting the AFS. If `True`, the likelihood is computed for an optimally rescaled FS. If `False`, the likelihood is computed for a fixed scaling of the FS found by theta=4*Ne*u and Leff
- **ns** (*list of ints, optional*) – The sample size, which is only needed if we are using the frequency spectrum, as the sample size does not affect mean LD statistics.
- **statistics** (*list, optional*) – Defaults to `None`, which assumes that all statistics are present and in the conventional default order. If the data is missing some statistics, we must specify which statistics are present using the subset of statistic names given by `moments.LD.Util.moment_names(num_pops)`.
- **pass_Ne** (*bool, optional*) – Defaults to `False`. If `True`, the demographic model includes `Ne` as a parameter (in the final position of input parameters).
- **maxiter** (*int*) – Defaults to None. Maximum number of iterations to perform.
- **maxfun** (*int*) – Defaults to None. Maximum number of function evaluations to make.

moments.LD.Inference.**optimize_log_lbfgsb**(*p0*, *data*, *model_func*, *rs=None*, *theta=None*, *u=2e-08*, *Ne=None*, *lower_bound=None*, *upper_bound=None*, *verbose=0*, *flush_delay=0.5*, *normalization=0*, *func_args=[]*, *func_kwargs={}*, *fixed_params=None*, *use_afs=False*, *Leff=None*, *multinom=False*, *ns=None*, *statistics=None*, *pass_Ne=False*, *spread=None*, *maxiter=40000*, *epsilon=0.001*, *pgtol=1e-05*)

Optimize (using the log of) the parameters using the modified Powell's method, which optimizes slices of parameter space sequentially. Initial parameters `p0`, the data `[means, varcovs]`, the demographic `model_func`, and `rs` to specify recombination bin edges are required. `Ne` must either be specified as a keyword argument or is included as the *last* parameter in `p0`.

It is best at burrowing down a single minimum. This method is better than optimize_log if the optimum lies at one or more of the parameter bounds. However, if your optimum is not on the bounds, this method may be much slower.

Because this works in log(params), it cannot explore values of params < 0. It should also perform better when parameters range over scales.

The L-BFGS-B method was developed by Ciyou Zhu, Richard Byrd, and Jorge Nocedal. The algorithm is described in:

- R. H. Byrd, P. Lu and J. Nocedal. A Limited Memory Algorithm for Bound Constrained Optimization, (1995), SIAM Journal on Scientific and Statistical Computing , 16, 5, pp. 1190-1208.
- C. Zhu, R. H. Byrd and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization (1997), ACM Transactions on Mathematical Software, Vol 23, Num. 4, pp. 550-560.

Parameters

- **p0** (*list*) – The initial guess for demographic parameters, demography parameters plus (optionally) Ne.
- **data** (*list*) – The parsed data[means, varcovs, fs]. The frequency spectrum fs is optional, and used only if use_afs=True.
  - Means: The list of mean statistics within each bin (has length `len(rs)` or `len(rs) - 1` if using AFS). If we are not using the AFS, which is typical, the heterozygosity statistics come last.
  - varcovs: The list of varcov matrices matching the data in `means`.
- **model_func** (*list*) – The demographic model to compute statistics for a given rho. If we are using AFS, it's a list of the two models [LD func, AFS func]. If we're using LD stats alone, we pass a single LD model as a list: [LD func].
- **rs** (*list*) – The list of raw recombination rates defining bin edges.
- **theta** (*float, optional*) – The population scaled per base mutation rate (4*Ne*mu, not 4*Ne*mu*L).
- **u** (*float, optional*) – The raw per base mutation rate. Cannot be used with `theta`.
- **Ne** (*float, optional*) – The fixed effective population size to scale u and r. If `Ne` is a parameter to fit, it should be the last parameter in `p0`.
- **lower_bound** (*list, optional*) – Defaults to `None`. Constraints on the lower bounds during optimization. These are given as lists of the same length of the parameters.
- **upper_bound** (*list, optional*) – Defaults to `None`. Constraints on the upper bounds during optimization. These are given as lists of the same length of the parameters.
- **verbose** (*int, optional*) – If an integer greater than 0, prints updates of the optimization procedure at intervals given by that spacing.
- **func_args** (*list, optional*) – Additional arguments to be passed to `model_func`.

- **func_kwargs** (*dict, optional*) – Additional keyword arguments to be passed to model_func.
- **fixed_params** (*list, optional*) – Defaults to None. To fix some parameters, this should be a list of equal length as p0, with None for parameters to be fit and fixed values at corresponding indexes.
- **use_afs** (*bool, optional*) – Defaults to False. We can pass a model to compute the frequency spectrum and use that instead of heterozygosity statistics for single-locus data.
- **Leff** (*float, optional*) – The effective length of genome from which the fs was generated (only used if fitting to afs).
- **multinom** (*bool, optional*) – Only used if we are fitting the AFS. If True, the likelihood is computed for an optimally rescaled FS. If False, the likelihood is computed for a fixed scaling of the FS found by theta=4*Ne*u and Leff
- **ns** (*list of ints, optional*) – The sample size, which is only needed if we are using the frequency spectrum, as the sample size does not affect mean LD statistics.
- **statistics** (*list, optional*) – Defaults to None, which assumes that all statistics are present and in the conventional default order. If the data is missing some statistics, we must specify which statistics are present using the subset of statistic names given by moments.LD.Util.moment_names(num_pops).
- **pass_Ne** (*bool, optional*) – Defaults to False. If True, the demographic model includes Ne as a parameter (in the final position of input parameters).
- **maxiter** (*int*) – Defaults to 40,000. Maximum number of iterations to perform.
- **epsilon** – Step-size to use for finite-difference derivatives.
- **pgtol** (*float*) – Convergence criterion for optimization. For more info, see help(scipy.optimize.fmin_l_bfgs_b)

moments.LD.Inference.**optimize_log_powell**(*p0*, *data*, *model_func*, *rs=None*, *theta=None*, *u=2e-08*, *Ne=None*, *lower_bound=None*, *upper_bound=None*, *verbose=0*, *flush_delay=0.5*, *normalization=0*, *func_args=[]*, *func_kwargs={}*, *fixed_params=None*, *use_afs=False*, *Leff=None*, *multinom=False*, *ns=None*, *statistics=None*, *pass_Ne=False*, *spread=None*, *maxiter=None*, *maxfun=None*)

Optimize (using the log of) the parameters using the modified Powell's method, which optimizes slices of parameter space sequentially. Initial parameters p0, the data [means, varcovs], the demographic model_func, and rs to specify recombination bin edges are required. Ne must either be specified as a keyword argument or is included as the *last* parameter in p0.

Parameters

- **p0** (*list*) – The initial guess for demographic parameters, demography parameters plus (optionally) Ne.
- **data** (*list*) – The parsed data[means, varcovs, fs]. The frequency spectrum fs is optional, and used only if use_afs=True.
  - Means: The list of mean statistics within each bin (has length len(rs) or len(rs) - 1 if using AFS). If we are not using the AFS, which is typical, the heterozygosity statistics come last.
  - varcovs: The list of varcov matrices matching the data in means.

- **model_func** (*list*) – The demographic model to compute statistics for a given rho. If we are using AFS, it's a list of the two models [LD func, AFS func]. If we're using LD stats alone, we pass a single LD model as a list: [LD func].
- **rs** (*list*) – The list of raw recombination rates defining bin edges.
- **theta** (*float, optional*) – The population scaled per base mutation rate (4*Ne*mu, not 4*Ne*mu*L).
- **u** (*float, optional*) – The raw per base mutation rate. Cannot be used with `theta`.
- **Ne** (*float, optional*) – The fixed effective population size to scale u and r. If `Ne` is a parameter to fit, it should be the last parameter in `p0`.
- **lower_bound** (*list, optional*) – Defaults to `None`. Constraints on the lower bounds during optimization. These are given as lists of the same length of the parameters.
- **upper_bound** (*list, optional*) – Defaults to `None`. Constraints on the upper bounds during optimization. These are given as lists of the same length of the parameters.
- **verbose** (*int, optional*) – If an integer greater than 0, prints updates of the optimization procedure at intervals given by that spacing.
- **func_args** (*list, optional*) – Additional arguments to be passed to `model_func`.
- **func_kwargs** (*dict, optional*) – Additional keyword arguments to be passed to `model_func`.
- **fixed_params** (*list, optional*) – Defaults to `None`. To fix some parameters, this should be a list of equal length as `p0`, with `None` for parameters to be fit and fixed values at corresponding indexes.
- **use_afs** (*bool, optional*) – Defaults to `False`. We can pass a model to compute the frequency spectrum and use that instead of heterozygosity statistics for single-locus data.
- **Leff** (*float, optional*) – The effective length of genome from which the fs was generated (only used if fitting to afs).
- **multinom** (*bool, optional*) – Only used if we are fitting the AFS. If `True`, the likelihood is computed for an optimally rescaled FS. If `False`, the likelihood is computed for a fixed scaling of the FS found by theta=4*Ne*u and Leff
- **ns** (*list of ints, optional*) – The sample size, which is only needed if we are using the frequency spectrum, as the sample size does not affect mean LD statistics.
- **statistics** (*list, optional*) – Defaults to `None`, which assumes that all statistics are present and in the conventional default order. If the data is missing some statistics, we must specify which statistics are present using the subset of statistic names given by `moments.LD.Util.moment_names(num_pops)`.
- **pass_Ne** (*bool, optional*) – Defaults to `False`. If `True`, the demographic model includes `Ne` as a parameter (in the final position of input parameters).
- **maxiter** (*int*) – Defaults to None. Maximum number of iterations to perform.
- **maxfun** (*int*) – Defaults to None. Maximum number of function evaluations to make.

`moments.LD.Inference.`**remove_nonpresent_statistics**(*y*, *statistics=[[], []]*)

Removes data not found in the given set of statistics.

Parameters

- **y** (`LDstats` object.) – LD statistics.
- **statistics** – A list of lists for two and one locus statistics to keep.

moments.LD.Inference.**remove_normalized_data**(*means*, *varcovs*, *normalization=0*, *num_pops=1*, *statistics=None*)

Returns data means and covariance matrices with the normalizing statistics removed.

Parameters

- **means** (*list of arrays*) – List of means normalized statistics, where each entry is the full set of statistics for a given recombination distance.
- **varcovs** (*list of arrays*) – List of the corresponding variance covariance matrices.
- **normalization** (*int*) – The index of the normalizing population.
- **num_pops** (*int*) – The number of populations in the data set.

moments.LD.Inference.**remove_normalized_lds**(*y*, *normalization=0*)

Returns LD statistics with the normalizing statistic removed.

Parameters

- **y** (LDstats object) – An LDstats object that has been normalized to get σ_D^2-formatted statistics.
- **normalization** (*int*) – The index of the normalizing population.

moments.LD.Inference.**sigmaD2**(*y*, *normalization=0*)

Compute the σ_D^2 statistics normalizing by the heterozygosities in a given population.

Parameters

- **y** (LDstats object) – The input data.
- **normalization** (*int, optional*) – The index of the normalizing population (normalized by pi2_i_i_i_i and H_i_i), default set to 0.

16.5.2 Confidence intervals

Parameter uncertainties are computed using Godambe information, described in Coffman et al, MBE (2016). doi: https://doi.org/10.1093/molbev/msv255

If you use moments.LD.Godambe to compute parameter uncertainties, please cite that paper. This was first developed by Alec Coffman for computing uncertainties from inferences performed with dadi, modified here to handle LD decay curves.

moments.LD.Godambe.**FIM_uncert**(*model_func*, *p0*, *ms*, *vcs*, *log=False*, *eps=0.01*, *r_edges=None*, *normalization=0*, *pass_Ne=False*, *statistics=None*, *verbose=0*)

Parameter uncertainties from Fisher Information Matrix. This approach typically underestimates the size of the true confidence intervals, as it does not take into account linkage between loci that causes data to be non-independent.

Returns standard deviations of parameter values.

Parameters

- **model_func** – Model function
- **p0** – Best-fit parameters for model_func, with inferred Ne in last entry of parameter list.
- **ms** – See below..
- **vcs** – Original means and covariances of statistics from data. If statistics are not give, we remove the normalizing statistics. Otherwise, these need to be pared down so that the normalizing statistics are removed.

- **eps** – Fractional stepsize to use when taking finite-difference derivatives. Note that if eps*param is < 1e-12, then the step size for that parameter will simply be eps, to avoid numerical issues with small parameter perturbations.
- **log** – If True, assume log-normal distribution of parameters. Returned values are then the standard deviations of the *logs* of the parameter values, which can be interpreted as relative parameter uncertainties.
- **return_GIM** – If true, also return the full GIM.
- **r_edges** – The bin edges for LD statistics.
- **normalization** – The index of the population that we normalized by.
- **pass_Ne** – If True, Ne is a parameter in the model function, and by convention is the last entry in the parameters list. If False, Ne is only used to scale recombination rates.
- **statistics** – Statistics that we have included given as a list of lists: [ld_stats, h_stats]. If statistics is not given, we assume all statistics are included except for the normalizing statistic in each bin
- **verbose** (*int, optional*) – If an integer greater than 0, prints updates of the number of function calls and tested parameters at intervals given by that spacing.

moments.LD.Godambe.**GIM_uncert**(*model_func*, *all_boot*, *p0*, *ms*, *vcs*, *log=False*, *eps=0.01*, *return_GIM=False*, *r_edges=None*, *normalization=0*, *pass_Ne=False*, *statistics=None*, *verbose=0*)

Parameter uncertainties from Godambe Information Matrix (GIM). If you use this method, please cite Coffman et al., MBE (2016).

Returns standard deviations of parameter values.

Parameters

- **model_func** – Model function
- **all_boot** – List of bootstrap LD stat means [m0, m1, m2, ...]
- **p0** – Best-fit parameters for model_func, with inferred Ne in last entry of parameter list.
- **ms** – See below..
- **vcs** – Original means and covariances of statistics from data. If statistics are not give, we remove the normalizing statistics. Otherwise, these need to be pared down so that the normalizing statistics are removed.
- **eps** – Fractional stepsize to use when taking finite-difference derivatives. Note that if eps*param is < 1e-12, then the step size for that parameter will simply be eps, to avoid numerical issues with small parameter perturbations.
- **log** – If True, assume log-normal distribution of parameters. Returned values are then the standard deviations of the *logs* of the parameter values, which can be interpreted as relative parameter uncertainties.
- **return_GIM** – If true, also return the full GIM.
- **r_edges** – The bin edges for LD statistics.
- **normalization** – The index of the population that we normalized by.
- **pass_Ne** – If True, Ne is a parameter in the model function, and by convention is the last entry in the parameters list. If False, Ne is only used to scale recombination rates.

- **statistics** – Statistics that we have included given as a list of lists: [ld_stats, h_stats]. If statistics is not given, we assume all statistics are included except for the normalizing statistic in each bin
- **verbose** (*int, optional*) – If an integer greater than 0, prints updates of the number of function calls and tested parameters at intervals given by that spacing.

16.6 Plotting

Todo: These docs are still needed.

BIBLIOGRAPHY

[Jouganous2017] Jouganous, Julien, et al. "Inferring the joint demographic history of multiple populations: beyond the diffusion approximation." *Genetics* 206.3 (2017): 1549-1567.

[Krukov2021] Krukov, Ivan, and Simon Gravel. "Taming strong selection with large sample sizes." *bioRxiv* (2021), doi: 10.1101/2021.03.30.437711.

[Sawyer1992] Sawyer, Stanley A., and Daniel L. Hartl. "Population genetics of polymorphism and divergence." *Genetics* 132.4 (1992): 1161-1176.

[Coffman2016] Coffman, Alec J., et al. "Computationally efficient composite likelihood statistics for demographic inference." *Molecular biology and evolution* 33.2 (2016): 591-593.

[Hill1968] Hill, W. G., and Alan Robertson. "Linkage disequilibrium in finite populations." *Theoretical and applied genetics* 38.6 (1968): 226-231.

[Ragsdale2019] Ragsdale, Aaron P., and Simon Gravel. "Models of archaic admixture and recent history from two-locus statistics." *PLoS genetics* 15.6 (2019): e1008204.

[Ragsdale2020] Ragsdale, Aaron P., and Simon Gravel. "Unbiased estimation of linkage disequilibrium from unphased data." *Molecular Biology and Evolution* 37.3 (2020): 923-932.

[Ardlie2001] Ardlie, Kristin, et al. "Lower-than-expected linkage disequilibrium between tightly linked markers in humans suggests a role for gene conversion." *The American Journal of Human Genetics* 69.3 (2001): 582-589.

[Harris2014] Harris, Kelley, and Rasmus Nielsen. "Error-prone polymerase activity causes multinucleotide mutations in humans." *Genome research* 24.9 (2014): 1445-1454.

[Kelleher2016] Kelleher, Jerome, Alison M. Etheridge, and Gilean McVean. "Efficient coalescent simulation and genealogical analysis for large sample sizes." *PLoS computational biology* 12.5 (2016): e1004842.

[Coffman2016] Coffman, Alec J., et al. "Computationally efficient composite likelihood statistics for demographic inference." *Molecular biology and evolution* 33.2 (2016): 591-593.

[Gutenkunst2009] Gutenkunst, Ryan N., et al. "Inferring the joint demographic history of multiple populations from multidimensional SNP frequency data." *PLoS genet* 5.10 (2009): e1000695.

[Ragsdale2020] Ragsdale, Aaron P., et al. "Lessons learned from bugs in models of human history." *The American Journal of Human Genetics* 107.4 (2020): 583-588.

[Boyko] Boyko, Adam R., et al. "Assessing the evolutionary impact of amino acid mutations in the human genome." *PLoS Genetics* 4.5 (2008): e1000083.

[Karczewski] Karczewski, Konrad J., et al. "The mutational constraint spectrum quantified from variation in 141,456 humans." *Nature* 581.7809 (2020): 434-443.

[Keightley] Keightley, Peter D., and Adam Eyre-Walker. "Joint inference of the distribution of fitness effects of deleterious mutations and population demography based on nucleotide polymorphism frequencies." *Genetics* 177.4 (2007): 2251-2261.

[Kim] Kim, Bernard Y., Christian D. Huber, and Kirk E. Lohmueller. "Inference of the distribution of selection coefficients for new nonsynonymous mutations using large samples." *Genetics* 206.1 (2017): 345-361.

[Ragsdale] Ragsdale, Aaron P., et al. "Triallelic population genomics for inferring correlated fitness effects of same site nonsynonymous mutations." *Genetics* 203.1 (2016): 513-523.

[1000G] 1000 Genomes Project Consortium. "A global reference for human genetic variation." *Nature* 526.7571 (2015): 68-74.

[Garcia] Garcia, Jesse A., and Kirk E. Lohmueller. "Negative linkage disequilibrium between amino acid changing variants reveals interference among deleterious mutations in the human genome." *bioRxiv* (2020).

[Good] Good, Benjamin H. "Linkage disequilibrium between rare mutations." *Genetics* (2022).

[Hudson] Hudson, Richard R. "Two-locus sampling distributions and their application." *Genetics* 159.4 (2001): 1805-1817.

[Ohta] Ohta, Tomoko, and Motoo Kimura. "Linkage disequilibrium between two segregating nucleotide sites under the steady flux of mutations in a finite population." *Genetics* 68.4 (1971): 571.

[Ragsdale_Gutenkunst] Ragsdale, Aaron P. and Ryan N. Gutenkunst. "Inferring demographic history using two-locus statistics." *Genetics* 206.2 (2017): 1037-1048.

[Ragsdale_Gravel] Ragsdale, Aaron P. and Simon Gravel. "Models of archaic admixture and recent history from two-locus statistics." *PLoS Genetics* 15.8 (2019): e1008204.

[Sandler] Sandler, George, Stephen I. Wright, and Aneil F. Agrawal. "Using patterns of signed linkage disequilibria to test for epistasis in flies and plants." *bioRxiv* (2020).

[Sanjak] Sanjak, Jaleal S., Anthony D. Long, and Kevin R. Thornton. "A model of compound heterozygous, loss-of-function alleles is broadly consistent with observations from complex-disease GWAS datasets." PLoS genetics 13.1 (2017): e1006573.

[Sohail] Sohail, Mashaal, et al. "Negative selection in humans and fruit flies involves synergistic epistasis." *Science* 356.6337 (2017): 539-542.

PYTHON MODULE INDEX

m

INDEX

H

I

L

M

N

O

P

R

S

T

U

W

Z