Momens user manual

Corresponding to version 1.0

Contents

Example code

0 Foreword

Moments is a new efficient method to simulate frequency spectra. For convenience, we reused ∂a∂i's interface, just introducing our new computation engine. For that reason, both libraries are very similar in the way they are used so ∂a∂i users should switch easily to *Moments*. However, we denote a few slight differences and new convenience features that will be described in this manual. As the interfaces are similar, we just modified ∂a∂i's manual to be specific to *Moments*.

1 Introduction

Welcome to *Moments*!

Moments is a powerful software tool for simulating the joint frequency spectrum (FS) of genetic variation among multiple populations and employing the FS for population-genetic inference. An important aspect of *Moments* is its flexibility, particularly in model specification, but with that flexibility comes some complexity. *Moments* is not a GUI program, nor can *Moments* be run usefully with a single command at the command-line; using *Moments* requires at least rudimentary Python scripting. Luckily for us, Python is a beautiful and simple language. Together with a few examples, this manual will quickly get you productive with *Moments* even if you have no prior Python experience.

1.1 Helping us

As we do our own research, *Moments* is constantly improving. Our philosophy is to include in *Moments* any code we develop for our own projects that may useful to others. Similarly, if you develop *Moments*-related code that you think might be useful to others, please let us know so we can include it with the main distribution.

If you discover a bug in *Moments*, please submit an issue on the *Moments* repository: `https://bitbucket.org/jjouganous/moments`. Also, if you have particular needs that modification to *Moments* may fulfill, please contact the developers and we may be able to help.

2 Suggested workflow

One of Python's major strengths is its interactive nature. This is very useful in the exploratory stages of a project: for examining data and testing models. If you intend to use *Moments*'s plotting commands, which rely on `matplotlib`, they you'll almost certainly want to install IPython, an enhanced Python shell that fixes several difficulties with interactive plotting using `matplotlib`.

My preferred workflow involves one window editing a Python script (e.g. `script.py`) and another running an IPython session (started as `ipython -pylab`). In the IPython session I can interactively use *Moments*, while I record my work in `script.py`. IPython's

%run script.py magic command lets me apply changes I've made to script.py to my interactive session. (Note that you will need to reload other Python modules used by your script if you change them.) Once I'm sure I've defined my model correctly and have a useful script, I run that from the command line (python script.py) for extended optimizations and other long computations.

Note that to access *Moments*'s functions, you will need to import moments at the start of your script or interactive session.

If you are comfortable with Matlab, this workflow should seem very familiar. Moreover the numpy, scipy, and matplotlib packages replicate much of Matlab's functionality.

3 Importing data

Moments represents frequency spectra using moments.Spectrum objects. As described in section 4, Spectrum objects are subclassed from numpy.masked_array and thus can be constructed similarly The most basic way to create a Spectrum is manually:

```
fs = moments.Spectrum([0,100,20,10,1,0])
```

This creates a Spectrum object representing the FS from a single population, from which we have 5 samples. (The first and last entries in fs correspond to mutations observed in zero or all samples. These are thus not polymorphisms, and by default *Moments* masks out those entries so they are ignored.)

For nontrivial data sets, entering the FS is manually infeasible, so we will focus on automatic methods of generating a Spectrum object. The most direct way is to load a pre-generated FS from a file, using

```
fs = moments.Spectrum.from_file(filename)
```

The appropriate file format is detailed in the next section.

3.1 Frequency spectrum file format

Moments uses a simple file format for storing the FS. Each file begins with any number of comment lines beginning with #. The first non-comment line contains P integers giving the dimensions of the FS array, where P is the number of populations represented. For a FS representing data from 4x4x2 samples, this would be 5 5 3. (Each dimension is one larger than the number of samples, because the number of observations can range, for example, from 0 to 4 if there are 4 samples, for a total of 5 possibilities.) On the same line, the string folded or unfolded denoting whether or not the stored FS is folded.

The actual data is stored in a single line listing all the FS elements separated by spaces, in the order fs[0,0,0] fs[0,0,1] fs[0,0,2]. . . fs[0,1,0] fs[0,1,1]. . . . This is followed by a single line giving the elements of the mask in the same order as the data, with 1 indicating masked and 0 indicating unmasked.

The file corresponding to the Spectrum fs can be written using the command:

```
fs.to_file(filename)
```

```
Human Chimp Allele1 YRI  CEU  Allele2 YRI CEU Gene  Position
ACG   ATG   C       29   24   T       1   0   abcb1 289
CCT   CCT   C       29   23   G       3   2   abcb1 345
```

Listing 1: Example of SNP file format

3.2 SNP data format

As a convenience, *Moments* includes several methods for generating frequency spectra directly from SNP data. That relevant SNP file format is described here. A large example can be found in the `examples/fs_from_data/data.txt` file included with the *Moments* source distribution, and a small example is shown in Listing 1.

The data file begins with any number of comment lines that being with `#`. The first parsed line is a column header line. Whitespace is used to separate entries within the table, so no spaces are allowed within any entry. Individual rows make be commented out using `#`.

The first column contains the in-group reference sequence at that SNP, including the flanking bases. If the flanking bases are unknown, they can be denoted by `-`. The header label is arbitrary.

The second column contains the aligned outgroup reference sequence at that SNP, including the flanking bases. Unknown entries can be denoted by `-`. The header label is arbitrary.

The third column gives the first segregating allele. The column header must be exactly `Allele1`.

Then follows an arbitrary number of columns, one for each population, each giving the number of times Allele1 was observed in that population. The header for each column should be the population identifier.

The next column gives the second segregating allele. The column header must be exactly `Allele2`.

Then follows one column for each population, each giving the number of times Allele2 was observed in that population. The header for each column should be the population identifier, and the columns should be in the same order as for the Allele1 entries.

Then follows an arbitrary number of columns which will be concatenated with `_` to assign a label for each SNP.

The `Allele1` and `Allele2` headers must be exactly those values because the number of columns between those two is used to infer the number of populations in the file.

3.3 SNP data methods

The method `Misc.make_data_dict` reads the above SNP file format to generate a Python data dictionary describing the data:

```
dd = Misc.make_data_dict(filename)
```

From this dictionary, the method `Spectrum.from_data_dict` can be used to create a `Spectrum`.

```
fs = Spectrum.from_data_dict(dd, pop_ids=[’YRI’, ’CEU’],
                                    projections=[10, 12],
                                    polarized=True)
```

The `pop_ids` argument specifies which populations to use to create the FS, and their order. `projections` denotes the population sample sizes for the resulting FS. (Recall that for a diploid organism, assuming random mating, we get two samples from each individual.) Note that the total number of calls to Allele1 and Allele2 in a given population need not be the same for each SNP. When constructing the Spectrum each SNP will be projected down to the requested number of samples in each population. (Note that SNPs cannot be projected up, so SNPs without enough calls in any population will be ignored.) `polarized` specifies whether *Moments* should use outgroup information to polarize the SNPs. If `polarized=True`, SNPs without outgroup information, or with that information `-` will be ignored. If `polarized=False`, outgroup information will be ignored and the resulting `Spectrum` will be folded.

If your data have missing calls for some individuals, projecting down to a smaller sample size will increase the number of SNPs you can use for analysis. On the other hand, some fraction of the SNPs will now project down to frequency 0, and thus be uniformative. As a rule of thumb, we often choose our projection to maximize the number of segregating sites in our final fs (assessed via `fs.S()`), although we have not formally tested whether this maximizes statistical power.

The method `Spectrum.from_data_dict_corrected` polarizes the SNPs using outgroup information and applies a statistical correction for multiple mutations described by Hernandez et al. [?]. Any SNPs without full trinucleotide ingroup and outgroup sequences will be ignored, as well as SNPs in which the flanking bases are not conserved between ingroup and outgroup, or in which the outgroup allele is not one of the segregating alleles. The correction uses the expected number of substitutions per site, the trinucleotide mutation rate matrix, and a stationary trinucleotide distribution. These are summarized in a table of misidentification probabilities that can be calculated using `Misc.make_fux_table`. (It should also be possible to develop a correction using only the single-site transition matrix. If this would be helpful, please contact the developers of *Moments*.)

Moreover, *Moments* introduces a new method to build the data dictionary directly from vcf files:

```
dd = Misc.make_data_dict_vcf(vcf_filename, ['YRI', 'CEU']).
```

4 Manipulating spectra

Frequency spectra are stored in `moments.Spectrum` objects. Computationally, these are a subclass of `numpy.masked_array`, so most of the standard array manipulation techniques can be used. (In the examples here, I will typically be considering two-dimensional spectra, although all these features apply to higher-dimensional spectra as well.)

You can do arithmetic with `Spectrum` objects:

```
fs3 = fs1 + fs2
fs2 = fs1 * 2
```

Note that most operations involving two `Spectrum` objects only make sense if they correspond to data with the same sample sizes.

Standard indexing and slicing operations work as well. For example, to access the counts corresponding to 3 observations in population 1 and 5 observations in population 2, simply

```
counts = fs[3,5]
```

More complicated slices are also possible. The slice notation : indicates taking all corresponding entries. For example, to access the slice of the `Spectrum` corresponding to entries with 2 derived allele observations in population 2, take

```
slice = fs[:,2]
```

4.1 Summary statistics

The frequency spectrum encompasses many common summary statistics, and *Moments* provides methods to calculate them from `Spectrum` objects.

4.1.1 Single-population statistics

Watterson's theta can be calculated as

```
thetaW = fs.Watterson_theta()
```

The expected heterozygosity π assuming random mating is

```
pi = fs.pi()
```

Tajima's D is

```
D = fs.Tajima_D()
```

4.1.2 Multi-population statistics

The number of segregating sites S is simply the sum of all entries in the FS (except for the absent-in-all and derived-in-all entries). This can be calculated as

```
S = fs.S()
```

Wright's F_{ST} can be calculated as

```
Fst = fs.Fst()
```

This estimate of Fst assumes random mating, because the FS does not store heterozygote. Calculation is by the method of Weir and Cockerham [?]. 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.

4.2 Folding

By default, *Moments* considers the data in the `Spectrum` to be polarized, i.e. that the ancestral state of each variant is known. In some cases, however, this may not be possible, and the FS must be *folded*, indicating that only the minor allele frequency is known. To fold a `Spectrum` object, simply

```
folded = fs.fold()
```

The Spectrum object will record the fact that it has been folded, so that the likelihood and optimization machinery can automatically fold model spectra when the data are folded.

4.3 Masking

Finally, Spectrum arrays are *masked*, i.e. certain entries can be set to be ignored. Most typically, the ignored entries are the two corners: [0,0] and [n1,n2], corresponding to variants observed in zero samples or in all samples. More sophisticated masking is possible, however. For example, if your calling algorithm is such that singletons in population 1 cannot be confidently called, you may want to ignore those entries of the FS in your analysis. To do so, simply

```
fs.mask[1,:] = True
```

Note that care must be taken when doing arithmetic with Spectrum objects that are masked in different ways.

4.4 Marginalizing

If one has a multidimensional Spectrum it may be useful to examine the marginalized Spectrum corresponding to a subset of populations. To do so, use the marginalize method. For example, consider a three-dimensional Spectrum consisting of data from populations A, B, and C. To consider the marginal two dimensional spectrum for populations A and C, we need marginalize over population B.

```
fsAC = fsABC.marginalize([1])
```

And to consider the marginal one-dimensional FS for population B, we marginalize over populations A and C.

```
fsB = fsABC.marginalize([0,2])
```

Note that the argument to marginalize is a list of dimensions to marginalize over, *indexed from 0.*

4.5 Projection

One can also project an FS down from a larger sample size to a smaller sample size. Implicitly, this involves averaging over all possible re-samplings of the larger sample size data. This is very often done in the case of missing data: if some sites could not be called in all individuals, one can set a lower bound on the number of successful calls necessary to include a SNP in the analysis; SNPs with more successful calls can then be projected down to that number of calls.

In *Moments*, this is implemented with the project method. For example, to project a two-dimensional FS down to sample sizes of 14 and 26, use

```
proj = fs.project([14,26])
```

4.6 Sampling

One can simulate Poisson sampling from an FS using the `sample` method.

```
sample = fs.sample()
```

Each entry in the `sample` output FS will have a Poisson number of counts, with mean given by the corresponding entry in `fs`. If all sites are completely unlinked, this is a proper parametric bootstrap from your FS.

4.7 Scrambling

Occasionally, one may wish to ask whether the FS really represents samples from two populations or rather subsamples from a single population. A rough check of this is to consider what the FS would look like if the population identifiers were scrambled amongst the individuals for whom you have data. The `scramble` method will do this.

```
scrambled = fs.scramble()
```

As an example, one could consider whether the FS for JPT and CHB shows evidence of differentiation between the two populations. Note that this is an informal test, and we have not developed the theory to assign statistical significance to the results. It is, nevertheless, a useful guide.

5 Specifying a model

A demographic model specifies population sizes and migration rates as a function of time, and it also includes discrete events such as population splittings. Unlike many coalescent-based simulators, demographic models in *Moments* are specified forward in time. Also note that all population sizes within a demographic model are specified relative to some reference population size N_{ref}.

One important subtlety is that within the demographic model function, by default the mutation parameter $\theta = 4N_{\text{ref}}\mu$ is set to 1. This is because the optimal θ for a given model and set of data is trivial to calculate, so *Moments* by default does this automatically in optimization (so-called “multinomial” optimization). See Section 5.3 for how to fix theta to a particular value in a demographic model.

5.1 Implementation

Demographic models are specified by defining a Python function. This function employs various methods defined by *Moments* to specify the demography.

When defining a demographic function the arguments must be specified in a particular order. The *first* argument must be a list of free parameters that will be optimized. The *second* argument (usually called `ns`) must be a list of sample sizes. Note that in *Moments*, contrary to ∂a∂i, we don’t have to specify a number of grid points as we do not use a classical frequency discretization. Any additional arguments (between the second and last) can be used to pass additional non-optimized parameters, using the func_args argument of the optimization methods. (See Listing ?? for an example.)

The demographic model function tracks the evolution of Φ the allele frequency spectrum.

All demographic models employed in *Moments* must begin with an equilibrium population of non-zero size. Φ for such a population can be generated using the method LinearSystem_1D.steady_state_1D.

Once we've created an initial Φ, we can begin to manipulate it. First, we can split Φ to simulate population splits. This can be done using the methods Manips.split_1D_to_2D, Manips.split_2D_to_3D_2, Manips.split_2D_to_3D_1 and so on. These methods take in an input Φ of up to four dimensions, and output a Φ of one greater dimension, corresponding to addition of a population. The added population is the last dimension of Φ. For example, if Manips.split_2D_to_3D_1 is used, population 1 will split into populations 1 and 3.

Along with these discrete manipulations of Φ, we have the continuous transformations as time passes, due to genetic drift at different population sizes or migration. This is handled by the integration method that handles spectra with up to five populations. This method takes three crucial parameters, nu, ns and T. T specifies the time of this integration and nu specifies the size of this(ese) population(s) relative to the reference during this time period and ns gives the size of the sample. If we consider several populations, nu and ns must be arrays of numpy arrays of the form nu = [nu1, nu2, nu3] where nui is the relative population size of population i.

The migration rates are gathered in matrix m, m[i, j] being the rate of migration *from pop 2 into pop 1.* It is equal to the fraction of individuals each generation in pop 1 that are new migrants from pop 2, times the $2N_{\text{ref}}$. Moreover, selection and dominance coefficients s and h are scalars if we are simulating a single population and arrays for multiple populations cases.

Note that for all these methods, the integration time T must be positive. To ensure this, it is best to define your time parameters as the *interval between* events rather than the absolute time of those events. For example, a size change happened a time Tsize before a population split Tsplit in the past.

Importantly, population sizes may be functions of time. This allows one to simulate exponential growth and other more complex scenarios. To do so, simply pass a function that takes a single argument (the time) and returns the given variable. The Python lambda expression is a convenient way to do this. For example, to simulate a single population growing exponentially from size nu0 to size nuF over a time T, one can do:

```
sts = moments.LinearSystem_1D.steady_state_1D(ns)
phi = moments.Spectrum(sts)
nu_func = lambda t: nu0 * (nuF/nu0)**(t/T)
phi.integrate(nu = nu_func, [ns], T=T)
```

Numerous examples are provided in Listings 2 through ??.

5.2 Units

The units *Moments* uses are slightly different than those used by some other programs, *ms* in particular.

In *Moments*, $\theta = 4N_{\text{ref}}\mu$, as is typical.

Times are given in units of $2N_{\text{ref}}$ generations. This differs from *ms*, where time is in units of $4N_{\text{ref}}$ generations. So to convert from a time in *Moments* to a time in *ms*, *divide* by 2.

Migration rates are given in units of $M_{ij} = 2N_{\text{ref}}m_{ij}$. Again, this differs from *ms*, where the scaling factor is $4N_{\text{ref}}$ generations. So to get equivalent migration (m_{ij}) in *ms* for a given rate in *Moments*, *multiply* by 2.

5.3 Fixed θ

If you wish to set a fixed value of $\theta = 4N_0\mu$ in your analysis, that information must be provided to the initial Φ creation function and the `Integration` functions. For an example, see Listing **??**, which defines a demographic model in which θ is fixed to be 137 for derived population 1. Derived pop 1 is thus the reference population for specifying all population sizes, so its size is set to 1 in the call to `Integration.two_pops`. When fixing θ, every `Integration` function must be told what the reference θ is, using the option `theta0`. In addition, the methods for creating an initial Φ distribution must be passed the appropriate value of θ using the `theta0` option.

```
def bottleneck(params, ns):
    nuB, nuF, T = params
    nu_func = lambda t: [nuB * numpy.exp(numpy.log(nuF/nuB) * t / T)]

    sts = moments.LinearSystem_1D.steady_state_1D(ns[0])
    fs = moments.Spectrum(sts)
    fs.integrate(nu_func, ns, T)

    return fs
```

Listing 2: **Bottleneck:** At time `TF` + `TB` in the past, an equilibrium population goes through a bottleneck of depth `nuB`, recovering to relative size `nuF`.

```
def growth(params, ns):
    nu, T = params

    nu_func = lambda t: [numpy.exp(numpy.log(nu) * t / T)]
    sts = moments.LinearSystem_1D.steady_state_1D(ns[0])
    fs = moments.Spectrum(sts)
    fs.integrate(nu_func, ns, T)

    return fs
```

Listing 3: **Exponential growth:** At time `T` in the past, an equilibrium population begins growing exponentially, reaching size `nu` at present.

```
def split_mig(params, ns):
    nu1, nu2, T, m = params
    sts = moments.LinearSystem_1D.steady_state_1D(ns[0] + ns[1])
    fs = moments.Spectrum(sts)
    fs = moments.Manips.split_1D_to_2D(fs, ns[0], ns[1])
    fs.integrate([nu1, nu2], ns, T, m = numpy.array([[0, m], [m, 0]]))

    return fs
```

Listing 4: **Split with migration:** At time `T` in the past, two population diverge from an equilibrium population, with relative sizes `nu1` and `nu2` and with symmetric migration at rate `m`.

```
def IM(params, ns):
    s, nu1, nu2, T, m12, m21 = params

    sts = moments.LinearSystem_1D.steady_state_1D(ns[0] + ns[1])
    fs = moments.Spectrum(sts)
    fs = moments.Manips.split_1D_to_2D(fs, ns[0], ns[1])

    nu1_func = lambda t: s * (nu1/s)**(t/T)
    nu2_func = lambda t: (1-s) * (nu2/(1-s))**(t/T)
    nu_func = lambda t: [nu1_func(t), nu2_func(t)]

    fs.integrate(nu_func, ns, T, dt_fac=0.01, m=numpy.array([[0, m12], [m2

    return fs
```

Listing 5: **Two-population isolation-with-migration:** The ancestral population splits into two, with a fraction `s` going into pop 1 and fraction `1-s` into pop 2. The populations then grow exponentially, with asymmetric migration allowed between them.

```
def OutOfAfrica((nuAf, nuB, nuEu0, nuEu, nuAs0, nuAs,
                mAfB, mAfEu, mAfAs, mEuAs, TAf, TB, TEuAs),
                (n1,n2,n3)):
    sts = moments.LinearSystem_1D.steady_state_1D(n1+n2+n3)
    fs = moments.Spectrum(sts)

    fs.integrate([nuAf], [n1+n2+n3], TAf, 0.05)

    fs = moments.Manips.split_1D_to_2D(fs, n1, n2+n3)

    mig1=numpy.array([[0, mAfB],[mAfB, 0]])
    fs.integrate([nuAf, nuB], [n1, n2+n3], TB, 0.05, m=mig1)

    fs = moments.Manips.split_2D_to_3D_2(fs, n2, n3)

    nuEu_func = lambda t: nuEu0*(nuEu/nuEu0)**(t/TEuAs)
    nuAs_func = lambda t: nuAs0*(nuAs/nuAs0)**(t/TEuAs)
    nu2 = lambda t: [nuAf, nuEu_func(t), nuAs_func(t)]
    mig2=numpy.array([[0, mAfEu, mAfAs],[mAfEu, 0, mEuAs],[mAfAs, mEuAs,

    fs.integrate(nu2, [n1, n2, n3], TEuAs, 0.05, m=mig2)

    return fs
```

Listing 6: **Out-of-Africa model from Gutenkunst (2009):** This model involves a size change in the ancestral population, a split, another split, and then exponential growth of populations 1 and 2.

6 Simulation and fitting

6.1 Likelihoods

Moments offers two complimentary ways of calculating the likelihood of the data FS given a model FS. The first is the Poisson approach, and the second is the multinomial approach.

In the Poisson approach, the likelihood is the product of Poisson likelihoods for each entry in the data FS, given an expected value from the model FS. This approach is relevant if θ_0 is an explicit parameter in your demographic function. Then the likelihood `ll` is

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

In the multinomial approach, before calculating the likelihood, *Moments* will calculate the optimal θ_0 for comparing model and data. (It turns out that this is just $\theta_0 = \sum \text{data} / \sum \text{model}$.) Because θ_0 is so trivial to estimate given the other parameters in the model, it is most efficient for it *not* to be an explicit parameter in the demographic function. Then the likelihood `ll` is

```
ll = moments.Inference.ll_multinomial(model, data)
```

The optimal θ_0 can be requested via

```
theta0 = moments.Inference.optimal_sfs_scaling(model, data)
```

6.2 Fitting

To find the maximum-likelihood model parameters for a given data set, *Moments* employs non-linear optimization. Several optimization methods are provided, as detailed in Section 6.4.

6.2.1 Parameter bounds

In their exploration, the optimization methods typically try a wide range of parameter values. For the methods that work in terms of log parameters, that range can be very wide indeed. As a consequence, the algorithms may sometimes try parameter values that are very far outside the feasible range and that cause *very* slow evaluation of the model FS. Thus, it is important to place upper and lower bounds on the values they may try. For divergence times and migration rates, large values cause slow evaluation, so it is okay to put the lower bound to 0 as long as the upper bound is kept reasonable. In our analyses, we often set the upper bound on times to be 10 and the upper bound on migration rates to be 20. For population sizes, very small sizes lead to very fast drift and consequently slow solution of the model equations; thus a non-zero lower bound is important, with the upper bound less so. In our analyses, we often set the lower bound on population sizes to be 10^{-2} or 10^{-3} (i.e. `1e-2` or `1e-3`).

If your fits often push the bounds of your parameter space (i.e., results are often at the bounds of one or more parameters), this indicates a problem. It may be that your bounds are too conservative, so try widening them. It may also be that your model is misspecified or that there are unaccounted biases in your data.

6.3 Fixing parameters

It is often useful to optimize only a subset of model parameters. A common example is doing likelihood-ratio tests on nested models. The optional argument `fixed_params` to the optimization methods facilitates this. As an example, if `fixed_params=[None,1.0,None,2.0]`, the first and third model parameters will be optimized, with the second and fourth parameters fixed to 1 and 2 respectively. Note that when using this option, a full length initial parameter set `p0` should be passed in.

6.4 Which optimizer should I use?

Moments provides a multitude of optimization algorithms, each of which performs best in particular circumstances.

The three most-general purpose routines are the BFGS methods implemented in `moments.Inference.op` and `moments.Inference.optimize_log` as well as the Powell's conjugate direction method: `moments.Inference.optimize_powell`. These perform a local search from a specified set of parameters, using an algorithm which attempts to estimate the curvature of the likelihood surface. However, these methods may have convergence problems if the maximum-likelihood parameters are at one or more of the parameter bounds.

Moments also implements two L-BFGS-B methods, `moments.Inference.optimize_lbfgsb` and `moments.Inference.optimize_log_lbfgsb`. These implement a variant of the BFGS method that deals much more efficiently with bounded parameter spaces. If your optimizations are often hitting the parameter bounds, try using these methods. Note that it is probably best to start with the vanilla BFGS methods, because the L-BFGS-B methods will always try parameter values at the bounds during the search. This can dramatically slow model fitting.

We also provide a simplex (a.k.a. amoeba) method in terms of log parameters, implemented in `moments.Inference.optimize_log_fmin`. This method does not use derivative information, so it may be more robust than the BFGS-based methods, but it is much slower.

Finally, there is a simple grid search, implemented in `moments.Inference.optimize_grid`.

Both BFGS and simplex are local search algorithms; thus they are efficient, but not guaranteed to find the global optimum. Thus, it is important to run several optimizations for each data set, starting from different initial parameters. If all goes well, multiple such runs will converge to the same set of parameters and likelihood, and this likelihood will the the highest found. This is strong evidence that you have indeed found the global optimum. To facilitate this, *Moments* provides a method `moments.Misc.perturb_params` that randomly perturbs the parameters passed in to generate a new initial point for the optimization.

7 Plotting

For your convenience, *Moments* provides several plotting methods. These all require installation of the Python library `matplotlib`.

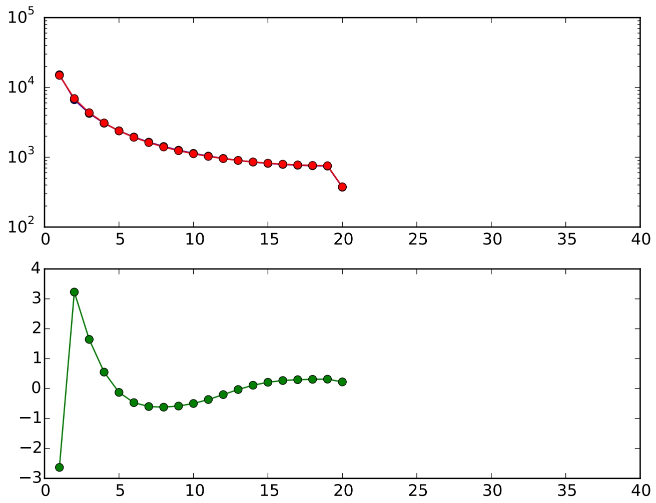

Figure 1: **1D model-data comparison plot:** In the top panel, the model is plotted in red and the data in blue. In the bottom panel, the residuals between model and data are plotted.

7.1 Essential matplotlib commands

To access additional, more general, methods for manipulating plots

```
import matplotlib.pyplot as pyplot
```

In particular, the method `pyplot.figure()` will create a new empty figure.

One quirk of `matplotlib` is that your plots may not show up immediately upon calling the plotting commands. If they don't, a call to `pyplot.show()` will pop them up. If you are not running in IPython, this will cause Python to block, so do not place it in scripts you run from the command-line, unless it is the last line.

7.2 1D comparison

`moments.Plotting.plot_1d_comp_Poisson` and `moments.Plotting.plot_1d_comp_multinomial` plot a comparison between a one-dimensional model and data FS. In the `_multinomial` method, the model is optimally scaled to match the data. The plot is illustrated in Fig. 1. The top plot shows the model and data frequency spectra, while the bottom shows the residuals between model and data. The bottom plot shows the residuals between model and data; a positive residuals means the model predicts too many SNPs in that entry. For an explanation of the residuals, see Section 7.6.

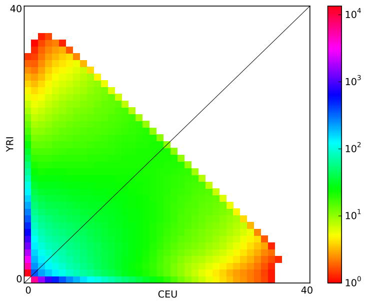

Figure 2: **2D FS plot:** Each entry in the FS is colored according to the logarithm of the number of variants within it.

7.3 2D spectra

`moments.Plotting.plot_single_2d_sfs` will plot a single two-dimensional frequency spectrum, as a logarithmic colormap. This is illustrated in Fig. 2, which is the result of

```
moments.Plotting.plot_single_2d_sfs(data, vmin=1)
```

Here `vmin` indicates the minimum value to plot, because in a logarithmic plot 0 in the FS maps to minus infinity, which causes great difficulty in plotting. Entires below the minimum (and masked entries) are plotted as white.

7.4 2D comparison

`moments.Plotting.plot_2d_comp_Poisson` and `moments.Plotting.plot_2d_comp_multinomial` plot comparisons between 2D models and data.

7.5 3D comparison

`moments.Plotting.plot_3d_comp_Poisson` and `moments.Plotting.plot_3d_comp_multinomial` plot comparisons between 3D models and data. The comparison is based on the 3 2D marginal spectra, a similar method exists to plot 4D spectra comparisons.

7.6 Residuals

The residuals are the properly normalized differences between model and data. Normalization is necessary, because the expected variance in each entry increase with the expected value of that entry. Two types of residuals are supported, Poisson and Anscombe.

The Poisson residual is simply

$$\text{residual} = (\text{model} - \text{data})/\sqrt{\text{model}}. \tag{1}$$

Note, however, that this residual is not normally distributed when the expected value (model entry) is small.

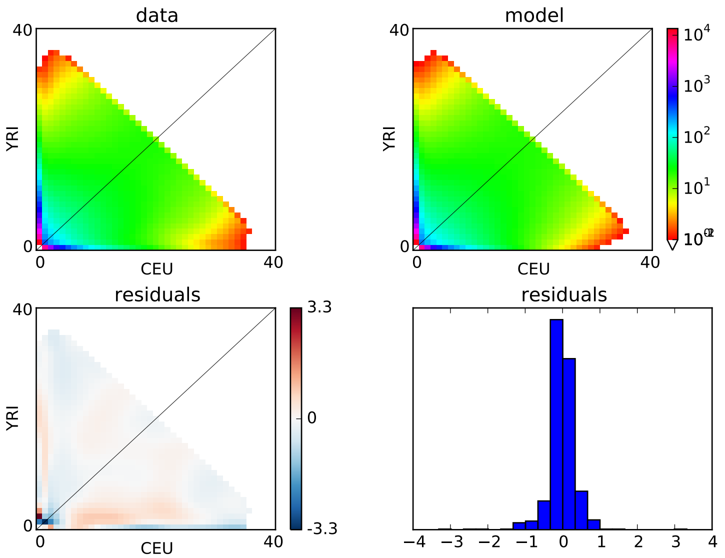

Figure 3: **2D model-data comparison plot:** The upper-left panel is the data, and the upper-right is the model. The lower two panels plot the residuals, and a histogram of the residuals.

The Anscombe residual is

$$\text{residual} = \frac{3}{2}\frac{(\text{model}^{\frac{2}{3}} - \text{model}^{-\frac{1}{3}}/9) - (\text{data}^{\frac{2}{3}} - \text{data}^{-\frac{-1}{3}}/9)}{\text{model}^{\frac{1}{6}}}. \tag{2}$$

These residuals are more normally distributed than the Poisson residuals when expected values are small [**?**].

7.7 Model plotting

8 Bootstrapping

Because *Moments*'s likelihood function treats all variants as independent, and they are often not, standard likelihood theory should not be used to estimate parameter uncertainties and significance levels for hypothesis tests. To do such tests, one can bootstrap. For estimating parameter uncertainties, one can use a nonparameteric bootstrap, i.e. sampling with replacement from independent units of your data (genes or chromosomes) to generate new data sets to fit. For hypothesis tests, the parametric bootstrap is preferred. This involves using a coalescent simulator (such as *ms*) to generate simulated data sets. Care must be taken to simulate the sequencing strategy as closely as possible. A method to generate bootstrapped frequency spectra accounting for linkage is provided: `moments.Misc.bootstrap`. This function takes in the data dictionary, the populations ID and the samples sizes and returns a set of bootstrapped frequency spectra.

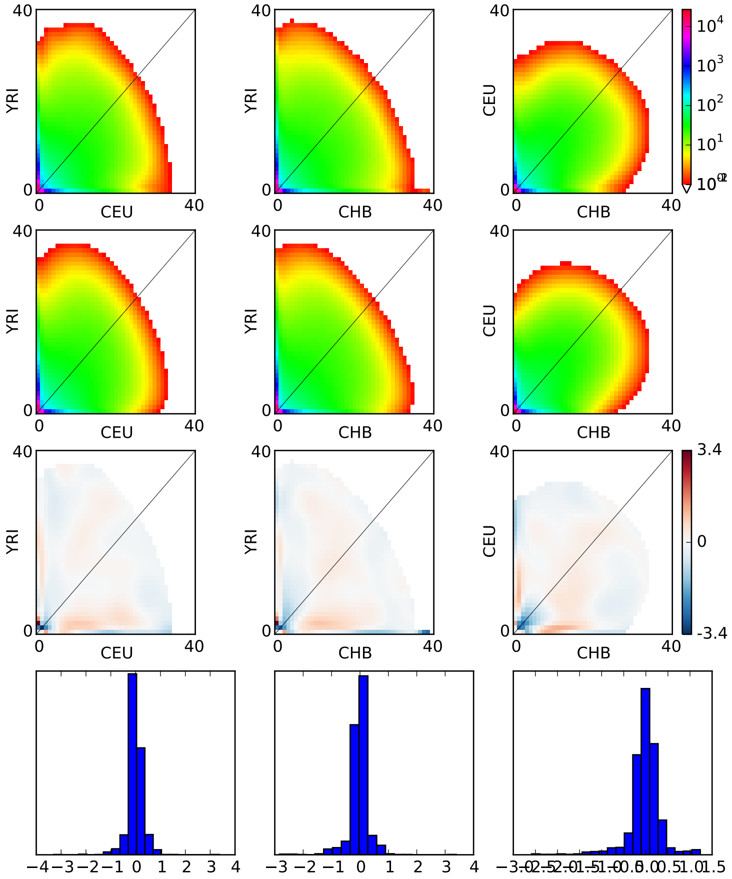

Figure 4: **3D model-data comparison plot:**

8.1 Interacting with *ms*

Moments provides several methods to ease interaction with *ms*. The method Spectrum.from_ms_file will generate an FS from *ms* output. The method Misc.ms_command will generate the command line for *ms* corresponding to a particular simulation. As an example:

```
import os

core = "-n 1 -n 2 -ej 0.3 2 1"
command = moments.Misc.ms_command(theta=1000, ns=(20,20), core, 1000,
                                  recomb=0.3)
ms_fs = moments.Spectrum.from_ms_file(os.popen(command))
```

Here the os.popen command lets us read the *ms* output straight from the command, without writing an intermediate file to disk. If you'd like to actually write the file, you could do

```
os.system("%s > temp.msout" % command)
ms_fs = moments.Spectrum.from_ms_file("temp.msout")
```

9 Uncertainty analysis

Moments can also perform uncertainty analysis using the Godambe Information Matrix (GIM), which is equivalent to the Fisher Information Matrix, but for composite likelihoods. The function call is

```
uncert = moments.Godambe.GIM_uncert(func_ex, grid_pts, all_boot,
                                    p0, data, log, multinom, eps).
```

Here func_ex is the model function, grid_pts is the set of grid points used in extrapolation, all_boot is a list containing bootstrapped data sets, p0 is the best-fit parameters, and data is the original data. If log = True, then uncertainties will be calculated for the logs of the parameters; these can be interpreted as relative uncertainties for the parameters themselves. If multinom = True, it is assumed that θ is not an explicit parameter of the model (this is the most common case). eps is the relative step size to use when taking numerical derivatives; the default value is often sufficient. The returned uncert is an array equal in length to p0, where each entry in uncert is the estimated standard deviation of the parameter it corresponds to in p0. If multinom = True, there will be one extra entry in uncert, corresponding to θ.

Using the GIM is often preferable to directly fitting the bootstrapped datasets, because such fitting is computationally time consuming. However, the GIM approach approximates parameter uncertainties as normal, which may not be a good approximation if they are large. To check this, one can evaluate the GIM uncertainties and compare them with the parameter values themselves. If the GIM uncertainties are large compared to the parameter values (for example, if a standard deviation is half the parameter value itself), then fitting the bootstrap data sets may be necessary to get accurate uncertainty estimates.

The moments.Godambe.FIM_uncert function calculates uncertainties using the Fisher Information Matrix, which is sufficient if your data are unlinked.

10 Likelihood ratio test

Using the Godambe Information Matrix, *Moments* can also perform hypothesis testing through an adjusted likelihood ratio test. The likelihood ratio test allows for comparison between two nested models, such that the simple model is a special case of the more complex model. The full likelihood ratio test statistic is equal to $D = 2(ll_c - ll_s)$, where ll_c and ll_s are the likelihoods of the complex and simple model, respectively. Model selection is then performed by comparing this test statistic to a χ^2 distribution with degrees of freedom equal to the difference in number of parameters between the simple and complex model. To perform likelihood ratio tests using composite likelihoods, a multiplicative adjustment to the likelihood ratio test statistic shown above is needed. *Moments* can calculate this adjustment, using the function.

```
adj = moments.Godambe.LRT_adjust(func_ex, grid_pts, all_boot, p0,
                                data, nested_indices, multinom=True, eps)
```

The parameters have the same meaning as for `Godambe.GIM_uncert`, where `func_ex` is the complex model function and `p0` is the best-fit parameters for the simple implemented in the complex model. The additional parameter `nested_indices` is a list that indicates which positions in the complex model arguments are fixed to create the simple model. For example, if the complex model parameters are $[T, \nu_1, \nu_2, m]$, and the simple model is no migration (so $m = 0$), then `nested_indices=[3]`. (Indices are numbered starting from zero.) The resulting adjusted D statistics is then $D_{adj} = adj \times 2(ll_c - ll_s)$.

In the simplest case of a single parameter on the interior of the complex parameter space, the null distribution for D_{adj} is χ^2 with 1 degree-of-freedom. If the a single parameter is on the boundary of the parameter space, the null distribution is $\frac{1}{2}\chi^2_0 + \frac{1}{2}\chi^2_1$. See [?] for an example of this. For convenience, *Moments* includes a function that computes the p-value given D.

```
p = sum_chi2_ppf(D, weights)
```

Here `D` is D_{adj} and `weights` records the weights in the sum-of-χ^2 distribution, beginning with zero degrees of freedom. For example, the case of a single parameter on the boundary would be `weights = (0.5, 0.5)`. For more complex scenarios, see [?].

11 Installation

11.1 Dependencies

Moments depends on a number of Python libraries. The absolute dependencies are

- Python, version ≥ 2.5 (but not Python 3)
- NumPy, version $\geq 1.2.0$
- SciPy, version $\geq 0.6.0$

It is also recommended that you install

- matplotlib, version ≥ 0.98.1
- IPython, version ≥ 0.10

For 3D plotting, it is also suggested that you install

- MayaVi2

The easiest way to obtain all these dependencies is to install Enthought's Python Distribution, which is free for academic use.

11.2 Installing from source

Moments can be easily installed from source code. In the `moments` directory, run `sudo python setup.py i` This will compile the Cython modules *Moments* uses and install those plus all *Moments* Python files in your Python installation's `site-packages` directory. A (growing) series of tests can be run in the `tests` directory, via `python run_tests.py`