dadi user manual

Corresponding to version 2.0.3

Contents

Example code

1 Introduction

Welcome to dadi!

dadi 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 dadi is its flexibility, particularly in model specification, but with that flexibility comes some complexity. dadi is not a GUI program, nor can dadi be run usefully with a single command at the command-line; using dadi 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 dadi even if you have no prior Python experience.

1.1 Getting help

Please join the `dadi-user` Google groups: `https://groups.google.com/group/dadi-user`. `dadi-user` is the preferred forum for asking questions and getting help. Before posting a question, take a moment to look through the `dadi-user` archives to see if your question has already been addressed. There are example scripts included in the source distribution: `https://bitbucket.org/gutenkunstlab/dadi/src/master/examples/`.

1.2 Helping us

As we do our own research, dadi is constantly improving. Our philosophy is to include in dadi any code we develop for our own projects that may useful to others. Similarly, if you develop dadi-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 have particular needs that modification to dadi 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 dadi'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 dadi, 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 dadi's functions, you will need to `import dadi` 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

dadi represents frequency spectra using `dadi.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 = dadi.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 dadi masks out those entries so they are ignored.)

For nontrivial data sets, entering the FS manually is 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 = dadi.Spectrum.from_file(filename)
```

The appropriate file format is detailed in the next section. We have also added function to generate the FS from a VCF file (3.4).

3.1 Frequency spectrum file format

dadi 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, dadi 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 dadi 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 dadi 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. [1]. 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 dadi.)

3.4 Frequency spectra from VCF files

In newer versions of dadi ($\geq$ 2.0.5), we have included functions for generating frequency spectra from VCF files directly. The main function for accomplishing this is `make_data_dict_vcf` in the `dadi.Misc` submodule. The function has two required arguments: (1) the name of the VCF file (can be gzipped [*.vcf.gz]) and (2) the name of a file describing how individuals map to populations. This second file is a plain-text, two-column file containing the individual names in column one and their respective populations in column two:

```
i0 pop0
i1 pop0
i2 pop0
...
iN pop2
```

Examples of these files can be found in the `examples/fs_from_data/` folder. Generating a frequency spectrum with these files can then be achieved through the creation of a data dictionary with the following two lines of code:

```
dd = dadi.Misc.make_data_dict_vcf("example.vcf.gz",
                                  "popfile.txt")
fs = dadi.Spectrum.from_data_dict(dd, ['pop0','pop1'],
                                      projections=[20,30],
                                      polarized=False)
```

The default version of the `make_data_dict_vcf` function will only include sites that don't have any missing data. Because of this, we have included an option to take a smaller subsample of individuals from a population at each site so that variants with less missing data than the specified subsampling size are not completely ignored. To specify how many individuals should be subsampled, the function takes an additional dictionary as an argument, where the dictionary simply maps the population names to the desired number of individuals to subsample.

```
# create the subsample dictionary
ss = {'pop0':5, 'pop1':10}
# pass it as an additional argument
dd = dadi.Misc.make_data_dict_vcf("example.vcf.gz",
                                  "popfile.txt",
                                  subsample=ss)
fs = dadi.Spectrum.from_data_dict(dd, ['pop0','pop1'],
                                      projections=[10,20],
                                      polarized=False)
```

Subsampling offers an alternative to down projecting your data that preserves individual genotypes. Projecting will consider sampled chromosomes/alleles as exchangeable across all individuals, which is usually OK for a randomly mating population. However, if you want to include inbreeding in your model (see section 13), then projecting will erase the signal of excess homozygosity that inbreeding creates by sampling chromosomes instead of individuals. When generating a frequency spectrum from a subsampled data dictionary, be sure to set the projections argument to 2 times the subsample size you specified for each population so that no down projecting is done.

4 Manipulating spectra

Frequency spectra are stored in `dadi.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 dadi 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 [2]. 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, dadi 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 dadi, 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 and admixture. Unlike many coalescent-based simulators, demographic models in dadi 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 dadi 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 dadi 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. The *last* argument (usually called `pts`) must be the number of grid points used in the calculation. 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 8 for an example.)

The demographic model function tracks the evolution of ϕ the density of mutations within the populations at given frequencies. This continuous density ϕ is approximated by its values on a grid of points, represented by the `numpy` array phi. Thus the first step in a demographic model is to specify that grid:

```
xx = dadi.Numerics.default_grid(pts)
```

Here `pts` is the number of grid points in each dimension for representing ϕ.

All demographic models employed in dadi must begin with an equilibrium population of non-zero size. ϕ for such a population can be generated using the method `PhiManip.phi_1D`. The most important parameter to this method is `nu`, which specifies the relative size of this ancestral population to the reference population. Most often, the reference population is the ancestral, so `nu` defaults to 1.

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 `PhiManip.phi_1D_to_2D`, `PhiManip.phi_2D_to_3D_split_1`, and `PhiManip.phi_2D_to_3D_split_2`. These methods take in an input ϕ of either one or two 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 `PhiManip.phi_2D_to_3D_split_1` is used, population 1 will split into populations 1 and 3. `phi_2D_to_3D_admix` is a more advanced version of the `2D_to_3D` methods that incorporates admixture. In this method, the proportions of pop 3 that are derived from pop 1 and pop 2 may be specified.

Direct admixture events can be specified using the methods `phi_2D_admix_1_into_2`, `phi_2D_admix_2_into_1`, `phi_3D_admix_1_and_2_into_3`, `phi_3D_admix_1_and_3_into_2`, and `phi_3D_admix_2_and_3_into_1`. These methods do not change the dimensionality of ϕ, but rather simulate discrete admixture events. For example, `phi_2D_admix_1_into_2` can be used to simulate a large discrete influx of individuals from pop 1 into pop 2. For example, this might model European (pop 1) admixture into indigenous Americans (pop 2). Note that the `PhiManip` methods for admixture can compromise the effectiveness of extrapolation for evaluating entries in the frequency spectrum corresponding to SNPs private to the recipient population. If your model involves admixture, you may obtain better accuracy by avoiding extrapolation and instead setting `pts_l` to be a list of length 1. Alternatively, if the admixture is the final event in your model, you can model admixture using the `admix_props` arguments for `Spectrum.from_phi`.

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 `Integration` methods, `Integration.one_pop`, `Integration.two_pops`, and `Integration.three_pops`. Each of these methods must be used with a `phi` of the appropriate dimensionality. `Integration.one_pop` takes two crucial parameters, `T` and `nu`. `T` specifies the time of this integration and `nu` specifies the size of this population relative to the reference during this time period. `Integration.two_pop` takes an integration time `T`, relative sizes for populations 1 and 2 `nu1` and `nu2`, and migration parameters `m12` and `m21`. The migration parameter `m12` specifies 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 $2N_{\text{ref}}$. `Integration.three_pops` is a straightforward extension of `two_pops` but now there are three population sizes and six migration parameters.

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 and migration rates (and selection coefficients) 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:

```
nu_func = lambda t: nu0 * (nuF/nu0)**(t/T)
phi = Integration.one_pop(nu=nu_func, T=T)
```

Numerous examples are provided in Listings 2 through 8.

5.2 Units

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

In dadi, $\theta = 4N_{\text{ref}}\mu$, 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 dadi 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 dadi, *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 7, 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.

5.4 Ancient sequences

If you have DNA samples from multiple timepoints, you can construct a frequency spectrum in which different axes correspond to samples from different timepoints. To support this in dadi, the `Integration.one_pop`, `two_pops`, and `three_pops` support `freeze` arguments.

When `True`, these arguments will "freeze" a particular population so that it no longer changes (although the relationship between SNPs in the frozen and unfrozen populations will change). Note that because time in dadi models is in genetic units, you need to be careful in how you specify the time of collection of your frozen sample. In this case, you likely want to run a model that explicitly includes θ as a parameter (see Section 5.3), so that you can convert from physical to genetic units within the model function.

```
def bottleneck(params, ns, pts):
    nuB,nuF,TB,TF = params
    xx = Numerics.default_grid(pts)

    phi = PhiManip.phi_1D(xx)
    phi = Integration.one_pop(phi, xx, TB, nuB)
    phi = Integration.one_pop(phi, xx, TF, nuF)

    fs = Spectrum.from_phi(phi, ns, (xx,))
    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, pts):
    nu,T = params

    xx = Numerics.default_grid(pts)
    phi = PhiManip.phi_1D(xx)

    nu_func = lambda t: numpy.exp(numpy.log(nu) * t/T)
    phi = Integration.one_pop(phi, xx, T, nu_func)

    fs = Spectrum.from_phi(phi, ns, (xx,))
    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, pts):
    nu1,nu2,T,m = params

    xx = Numerics.default_grid(pts)

    phi = PhiManip.phi_1D(xx)
    phi = PhiManip.phi_1D_to_2D(xx, phi)

    phi = Integration.two_pops(phi, xx, T, nu1, nu2, m12=m, m21=m)

    fs = Spectrum.from_phi(phi, ns, (xx,xx))
    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, pts):
    s,nu1,nu2,T,m12,m21 = params

    xx = Numerics.default_grid(pts)

    phi = PhiManip.phi_1D(xx)
    phi = PhiManip.phi_1D_to_2D(xx, phi)

    nu1_func = lambda t: s * (nu1/s)**(t/T)
    nu2_func = lambda t: (1-s) * (nu2/(1-s))**(t/T)
    phi = Integration.two_pops(phi, xx, T, nu1_func, nu2_func,
                               m12=m12, m21=m21)

    fs = Spectrum.from_phi(phi, ns, (xx,xx))
    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.

```
from dadi import Numerics, PhiManip, Integration, Spectrum

def OutOfAfrica(params, ns, pts):
    nuAf, nuB, nuEu0, nuEu, nuAs0, nuAs,
                  mAfB, mAfEu, mAfAs, mEuAs, TAf, TB, TEuAs = params
    xx = Numerics.default_grid(pts)

    phi = PhiManip.phi_1D(xx)
    phi = Integration.one_pop(phi, xx, TAf, nu=nuAf)

    phi = PhiManip.phi_1D_to_2D(xx, phi)
    phi = Integration.two_pops(phi, xx, TB, nu1=nuAf, nu2=nuB,
                               m12=mAfB, m21=mAfB)

    phi = PhiManip.phi_2D_to_3D_split_2(xx, phi)

    nuEu_func = lambda t: nuEu0*(nuEu/nuEu0)**(t/TEuAs)
    nuAs_func = lambda t: nuAs0*(nuAs/nuAs0)**(t/TEuAs)
    phi = Integration.three_pops(phi, xx, TEuAs, nu1=nuAf,
                                 nu2=nuEu_func, nu3=nuAs_func,
                                 m12=mAfEu, m13=mAfAs, m21=mAfEu,
                                 m23=mEuAs, m31=mAfAs, m32=mEuAs)

    fs = Spectrum.from_phi(phi, (n1,n2,n3), (xx,xx,xx))
    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. (The `from dadi import` line imports those modules from the `dadi` namespace into the local namespace, so we don't have to type `dadi.` to access them.)

```
def fixed_theta(params, ns, pts):
    nuA, nu2, T = params
    theta1 = 137

    xx = dadi.Numerics.default_grid(pts)

    phi = dadi.PhiManip.phi_1D(xx, nu=nuA, theta0=theta1)
    phi = dadi.PhiManip.phi_1D_to_2D(xx, phi)
    phi = dadi.Integration.two_pops(phi, xx, T, nu1=1, nu2=nu2,
                                    theta0=theta1)

    fs = dadi.Spectrum.from_phi(phi, ns, (xx,xx))
    return fs
```

Listing 7: **Fixed θ:** A split demographic model function with a fixed value of θ=137 for derived population 1. The free parameters are the sizes of the ancestral pop, `nuA`, and derived pop 2, `nu2`, (relative to derived pop 1), along with the divergence time `T` between the two derived pops.

```
from dadi import Numerics, PhiManip, Integration, Spectrum

def NewWorld(params, ns, fixed_params, pts):
    nuEu0, nuEu, nuAs0, nuAs, nuMx0, nuMx,
              mEuAs, TEuAs, TMx, fEuMx = params
    theta0, nuAf, nuB, mAfB, mAfEu, mAfAs, TAf, TB = fixed_params
    xx = Numerics.default_grid(pts)

    phi = PhiManip.phi_1D(xx)
    phi = Integration.one_pop(phi, xx, TAf, nu=nuAf)

    phi = PhiManip.phi_1D_to_2D(xx, phi)
    phi = Integration.two_pops(phi, xx, TB, nu1=nuAf, nu2=nuB,
                               m12=mAfB, m21=mAfB)

    # Integrate out the YRI population
    phi = Numerics.trapz(phi, xx, axis=0)

    phi = PhiManip.phi_1D_to_2D(xx, phi)
    nuEu_func = lambda t: nuEu0*(nuEu/nuEu0)**(t/(TEuAs+TMx))
    nuAs_func = lambda t: nuAs0*(nuAs/nuAs0)**(t/(TEuAs+TMx))
    phi = Integration.two_pops(phi, xx, TEuAs,
                               nu1=nuEu_func, nu2=nuAs_func,
                               m12=mEuAs, m21=mEuAs)
    phi = PhiManip.phi_2D_to_3D_split_2(xx, phi)

    # Initial population sizes for this stretch of integration
    nuEu0 = nuEu_func(TEuAs)
    nuAs0 = nuAs_func(TEuAs)
    nuEu_func = lambda t: nuEu0*(nuEu/nuEu0)**(t/TMx)
    nuAs_func = lambda t: nuAs0*(nuAs/nuAs0)**(t/TMx)
    nuMx_func = lambda t: nuMx0*(nuMx/nuMx0)**(t/TMx)
    phi = Integration.three_pops(phi, xx, TMx,
                                 nu1=nuEu_func, nu2=nuAs_func,
                                 nu3=nuMx_func,
                                 m12=mEuAs, m21=mEuAs,
                                 m23=mAsMx, m32=mAsMx)
    phi = PhiManip.phi_3D_admix_1_and_2_into_3(phi, fEuMx, 0,
                                               xx,xx,xx)

    fs = Spectrum.from_phi(phi, ns, (xx,xx,xx))
    # Apply our theta0. (All previous methods default to
    # theta0=1.)
    return theta0*fs
```

Listing 8: **Settlement-of-New-World model from Gutenkunst (2009):** Because dadi is limited to 3 simultaneous populations, we need to integrate out the African population, using `Numerics.trapz`. This model also employs a fixed θ, and ancillary parameters passed in using the third argument.

6 Simulation and fitting

6.1 Grid sizes and extrapolation

To simulate the frequency spectrum, dadi solves a partial differential equation, approximating the solution using a grid of points in population frequency space (the `phi` array). Importantly, a single evaluation of the frequency spectrum with a fixed grid size is apt to be inaccurate, because computational limits mean the grid must be relatively coarse. To overcome this, dadi solves the problem at a series (typically 3) of grid sizes and extrapolates to an infinitely fine grid. To transform the demographic model function you have created (call it `my_demo_func`) into a function that does this extrapolation, wrap it using a call to `Numerics.make_extrap_func`, e.g.:

```
my_extrap_func = Numerics.make_extrap_func(my_demo_func)
```

Having done this, the final argument to `my_extrap_func` is now a *sequence* of grid sizes, which will be used for extrapolation. In our experience, good results are obtained by setting the smallest grid size slightly larger than the largest population sample size. For example, if you have sample sizes of 16, 24, and 12 samples in the three populations you're working with, a good choice of grid sizes is probably `pts_l = [40,50,60]`. This can be altered depending on your usage. For example, if you are fitting a complex slow model, it may speed up the analysis considerably to first run an optimization at small grid sizes (even less than the maximum number of samples). This should get your parameter values approximately correct. They can be refined by running another optimization with a finer grid.

A simulated frequency spectrum is thus obtained by calling

```
model = my_extrap_func(params, ns, pts_l)
```

Here `ns` is the sequence of sample sizes for the populations in the model, `params` is the model parameters, and `pts_l` is the grid sizes.

6.1.1 Grid choice

As of version 1.5.0, the default grid in dadi has points exponentially clustered toward $x = 0$ and $x = 1$. This grid was suggested by Simon Gravel. The parameter `crwd` controls how closely grid points crowd the endpoints of the interval.

We have performed some empirical investigations of the best value for `crwd`, although these results cannot be considered definitive. We ran simulations for a variety of models and parameter values for a variety of sample sizes. Denoting the largest sample size as `n`, we asked which value of `crwd` yielded the most accurate FS with `pts_l = [n, n+10, n+20]`. Results are shown in Fig. 1. It is evident that the best value for `crwd` is lower for smaller sample sizes. The red lines are empirical functions which approximate the optimum. These are implemented in `Numerics.estimate_best_exp_grid_crwd`. Fig. 2 demonstrates that the optimum value of `crwd` doesn't depend strongly on the number of grid points used for integration. Unless you need absolute top performance, the default value of `crwd=8` will probably be sufficient.

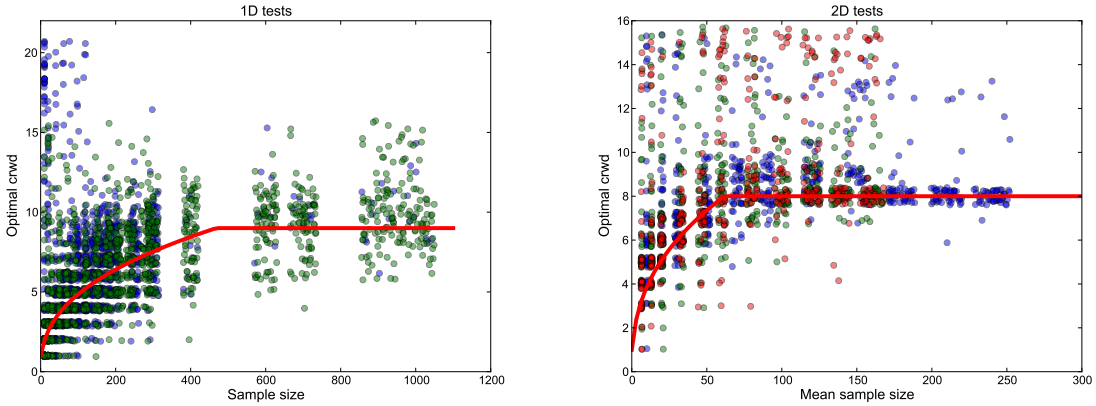

Figure 1: **Empirical optimum values for `crwd`:** Each point represents the optimum value of `crwd` for a given model with a particular random choice of parameters.

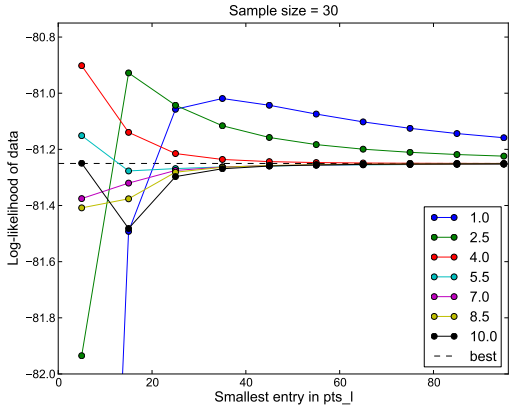

Figure 2: **Consistency of optimum `crwd` value:** For a one-dimensional system with 30 samples, the likelihood of a particular data set was calculated with `pts_l = [base, base+10, base+20]`, for varying `crwd` factors and values of `base`. In general, the optimum value of the `crwd` parameter does not depend on `base`.

6.2 Likelihoods

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

In the multinomial approach, before calculating the likelihood, dadi 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 = dadi.Inference.ll_multinomial(model, data)
```

The optimal θ_0 can be requested via

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

6.3 Fitting

To find the maximum-likelihood model parameters for a given data set, dadi employs nonlinear optimization. Several optimization methods are provided, as detailed in Section 6.5.

6.3.1 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.

6.4 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.

6.5 Which optimizer should I use?

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

The two most-general purpose routines are the BFGS methods implemented in `dadi.Inference.optimi` and `dadi.Inference.optimize_log`. 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.

dadi also implements two L-BFGS-B methods, `dadi.Inference.optimize_lbfgsb` and `dadi.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 `dadi.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 `dadi.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, dadi provides a method `dadi.Misc.perturb_params` that randomly perturbs the parameters passed in to generate a new initial point for the optimization.

7 Plotting

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

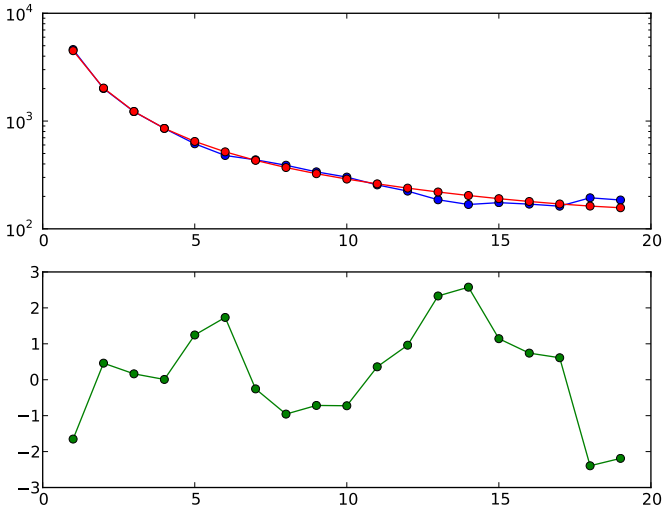

Figure 3: **1D model-data comparison plot:** In the top panel, the model is plotted in red and the data in blue. In the bottom panel, the residuals between model and data are plotted.

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

`dadi.Plotting.plot_1d_comp_Poisson` and `dadi.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. 3. 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 7.7.

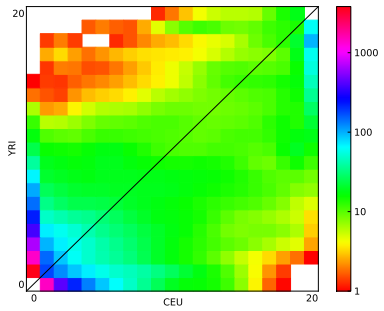

Figure 4: **2D FS plot:** Each entry in the FS is colored according to the logarithm of the number of variants within it.

7.3 2D spectra

`dadi.Plotting.plot_single_2d_sfs` will plot a single two-dimensional frequency spectrum, as a logarithmic colormap. This is illustrated in Fig. 4, which is the result of
`dadi.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

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

7.5 3D spectra

Unfortunately, nice portable 3D plotting is difficult in Python. We have developed a Mathematica script that will do such plotting (as in Fig. 2(A) of [3].) Please contact the authors `dadi-user` and we will send you a copy.

7.6 3D comparison

`dadi.Plotting.plot_3d_comp_Poisson` and `dadi.Plotting.plot_3d_comp_multinomial` plot comparisons between 3D models and data. The comparison is based on the 3 2D marginal spectra.

7.7 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

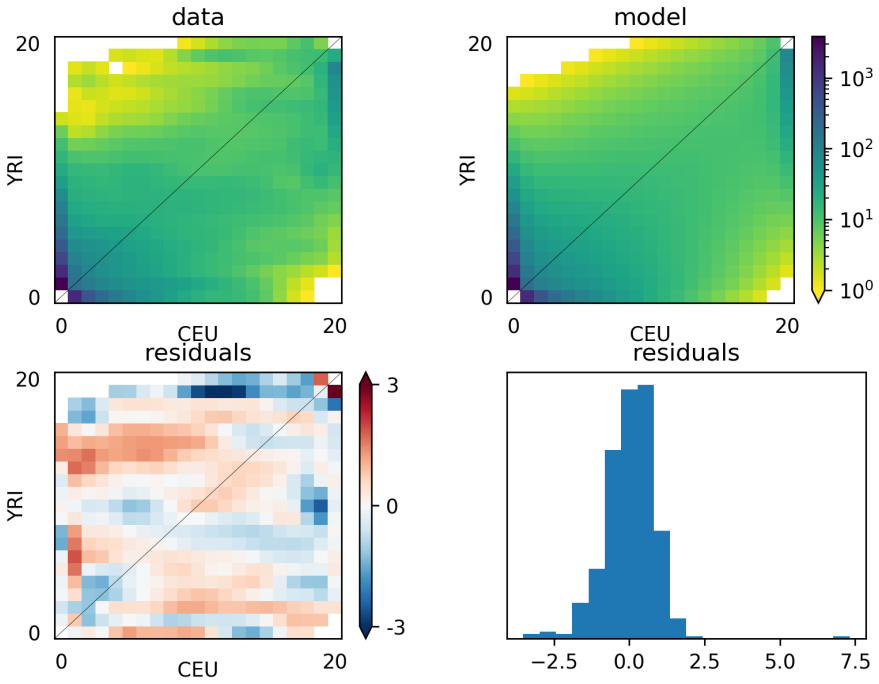

Figure 5: **2D model-data comparison plot:** The upper-left panel is the data, and the upper-right is the model. The lower two panels plot the residuals, and a histogram of the residuals.

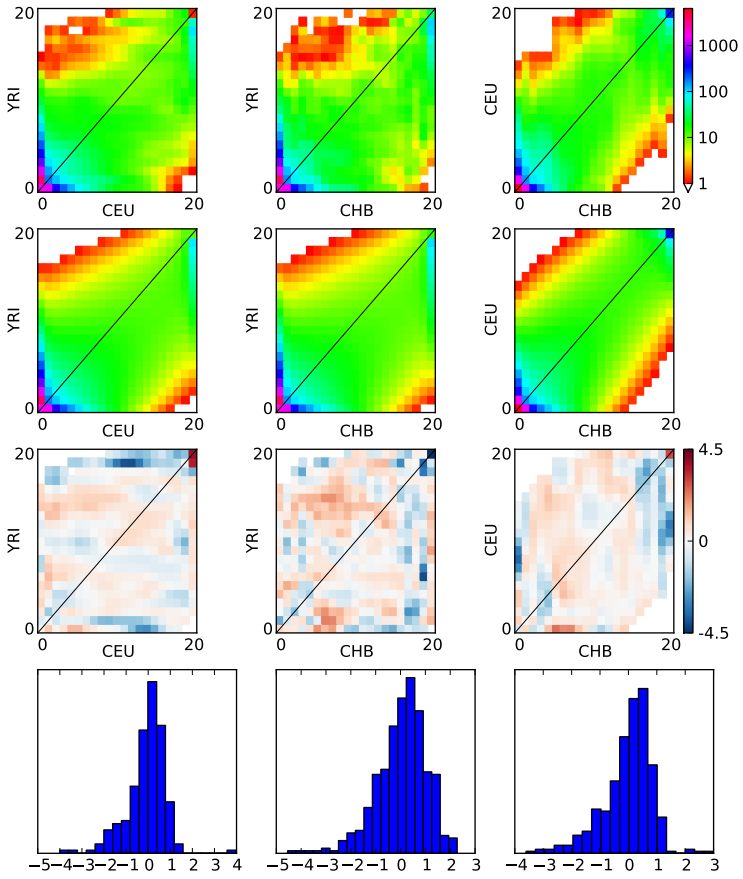

Figure 6: **3D model-data comparison plot:**

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.

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 [4].

8 Bootstrapping

Because dadi'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.

8.1 Interacting with *ms*

dadi 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 = dadi.Misc.ms_command(theta=1000, ns=(20,20), core, 1000,
                               recomb=0.3)
ms_fs = dadi.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 = dadi.Spectrum.from_ms_file("temp.msout")
```

9 Uncertainty analysis

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

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. If `return_GIM = True`, then the return value will be `(uncert,GIM)`, where `GIM` is the full Godambe Information Matrix, for use in propagating uncertainties.

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.

Parameter uncertainties for correlated parameters can also be determined using the GIM with uncertainty propagation techniques. An example of this would be if one wanted to know the uncertainty in the total time of a demographic model, $T_{total} = T1 + T2$ that contains two events occurring at times $T1$ and $T2$. If the variance for $T1$ and $T2$ are given by $\sigma^2_{T_1}$ and $\sigma^2_{T_2}$, with a covariance term between the two $\sigma_{T_1T_2}$, then the uncertainty in T_{total} is

$$\sigma_{T_{total}} = \sqrt{\sigma^2_{T_1} + \sigma^2_{T_2} + 2\sigma_{T_1T_2}}. \tag{3}$$

Another example where error propagation is necessary is when determining theta for an individual population, $\theta_A = \theta \times \nu_A$, from the overall theta, θ, and the relative population size ν_A. For variance in ν_A and θ given by $\sigma^2_{\nu_A}$ and σ^2_θ, respectively, and covariance between the two $\sigma_{\nu_A\theta}$, the equation for uncertainty in θ_A is

$$\sigma_{\theta_A} = \sqrt{\theta^2\sigma^2_{\nu_A} + {\nu_A}^2\sigma^2_\theta + 2\nu_A\theta\sigma_{\nu_A\theta}}. \tag{4}$$

The full GIM can be obtained from the `dadi.Godambe.GIM_uncert` function by setting `return_GIM=True`. Variances and covariances can be taken directly from the inverse of the

GIM (obtained by `numpy.linalg.inv(GIM)`, in which diagonal terms represent variance terms and off-diagonal terms represent covariance terms. For more complex scenarios, see [5].

The `dadi.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, dadi 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. dadi can calculate this adjustment, using the function.

```
adj = dadi.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. Results in [6] suggested that setting `p0` equal to the best-fit parameters from either the simple model or complex model yield similar adjustments, although the data in that paper were simulated under the simple model. When data was simulated under the complex model, it was found that evaluating the adjustment at the complex model parameterization was more powerful, yielding a more liberal adjustment than evaluating at the simple model parameterization. We suggest evaluating at the complex model parameterization, although evaluating at the simple model parameterization as well may offer additional insight and be preferable if you desire a more conservative estimate of the adjustment. 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 [6] for an example of this. For convenience, dadi 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 [7].

11 Triallelic spectra

The triallelic frequency spectrum is the distribution of frequencies of triallelic, instead of biallelic, SNPs. The triallelic spectrum stores the counts of observed alleles with given major and minor derived allele frequencies, where the major and minor derived alleles are those appearing at higher or lower frequency, resp. We use a `dadi.Spectrum` object for the triallelic spectrum as well, with entries for infeasible triallelic frequencies masked. The `dadi.Triallele` methods can handle selection at one or both derived alleles, and can produce expected frequency spectra under arbitrary, single-population demography. By folding a triallelic frequency spectrum, we assume that we do not know which derived allele arose first.

11.1 Built in models

In `dadi.Triallele.demographics.py`, you will find three pre-built demographic models: `equilibrium`, `two_epoch`, and `three_epoch`. The methods take demographic, selection, and integration parameters as inputs, as well as number of sampled individuals (`ns`), and number of grid points to use for integration (`pts`). For example, the parameters for the equilibrium model takes parameters `[sig1, sig2, theta1, theta2, misid, dt]`. The `sig` parameters are the selection coefficients for each derived allele, `theta` are the scaled mutation rates for each derived allele, `misid` is the probability of ancestral misidentification, and `dt` is the integration time step. For non-equilibrium demographies, those models also take population sizes `nu` relative to the ancestral population size, and times `T` for which the population is at size `nu`. An example can be found in the Examples subdirectory.

11.2 Faster triallele with Cython

The Triallele methods are written in Python; however, considerable speed-up can be achieved by generating some of the code in C. Some methods are written using Cython, and these can be installed by compiling the Cythonized code when dadi is built. To build the Cython extensions, use the flag `--cython_triallele` when installing dadi, by running
`sudo python setup.py install --cython_triallele`.
If for some reason installation fails to build the Cython modules, dadi can be installed without the Cythonized Triallele methods, and the Triallele methods will still be functional.

12 DFE Inference

Note: This section of the manual is adaptive from the fit∂a∂i manual written by Bernard Kim and Kirk Lohmueller. If you use this code, please be sure to cite their paper [8].

The code examples shown here are meant to work with the example dataset. For simplicity's sake, I have generated an example dataset with PReFerSIM [9]. Furthermore, we will work with a relatively small sample size and simple demographic model so that the examples can be worked through quickly on a laptop. Lastly, all the example code is provided in the `example.py` script as well as in this document.

Another important thing to note: ∂a∂i characterizes genotype fitnesses as: 1, $1+2sh$, and $1+2s$, where $1+2sh$ is the fitness of the heterozygote. Furthermore, the DFEs inferred are scaled in terms of the ancestral population size: $\gamma = 2N_A s$. This means that the selection coefficients must sometimes be rescaled, for instance when using the program SLiM [10].

12.1 Example dataset

The example dataset used in the example script was generated with forward simulations under the PRF model, with the simulation program PReFerSIM. Additionally, we will assume we know the true underlying demographic model rather than trying to fit one.

This dataset is summarized with a site frequency spectrum, has sample size $2n = 250$ (125 diploids), and is saved in the file `sample.sfs` file. It was generated with a single size change demography and an underlying gamma DFE. Specifically, a population of size $N = 10,000$ diploids expands to $20,000$ diploids 1000 generations ago and the gamma DFE has shape parameter 0.186 and scale parameter 686.7. This is the same gamma DFE that we inferred from the 1000 Genomes EUR dataset, but the scale parameter has been rescaled to the ancestral population size of 10,000 diploids. Finally, the amount of diversity in the sample dataset matches $\theta_{NS} = 4000 = 4N_A\mu L_{NS}$.

12.2 Demographic inference

Because the usage of ∂a∂i for demographic inference is extensively documented, it will not be discussed in detail here. In practice, we find that, as long as the demographic model that fits the synonymous sites reasonably well also works well for inference of the DFE.

Briefly, we fit a demographic model to synonymous sites, which are assumed to be evolving in a neutral or nearly neutral manner. We believe this accurately represents the effects of linked selection and population structure, and condition upon this demographic model to fit the DFE. However, note the assumption of neutral synonymous variants may not hold for species with large population sizes, since this will increase the efficacy of selection on mutations with small fitness effects.

Our sample dataset was generated with a two epoch (single size change) demography. We will assume we infer a 2-fold population expansion 0.05*$2N_A$ generations ago, where N_A is the ancestral population size. Therefore, the parameter vector is: `[nu, T]`.

```
if __name__ == '__main__':
    # Set demographic parameters and theta. This is usually inferred from
```

Again, we assume that the population scaled nonsynonymous mutation rate, $\theta_{NS} = 4,000$. In practice, we compute the synonymous mutation rate, θ_S, by using the multinomial likelihood to fit the demographic model. Because this method only fits the proportional

SFS, θ_S is estimated with the `dadi.Inference.optimal_sfs_scaling` method. Then, we multiply θ_S by 2.31 to get θ_{NS}, $\theta_S * 2.31 = \theta_{NS}$. Remember that our sample size is 125 diploids (250 chromosomes).

12.3 Pre-computing of the SFS for many γ

Next, we must generate frequency spectra for a range of gammas. The demographic function is modified to allow for a single selection coefficient. Here, each selection coefficient is scaled with the ancestral population size, $\gamma = 2N_A s$. In other words, if `gamma=0`, this function is the same as the original demographic function. This function is defined as `two_epoch` in `dadi.DFE.DemogSelModels.py`. Note the use of a Python decorator to easily define this as an extrapolating function.

```
    return Spectrum.from_phi(phi, ns, (xx,))

def two_epoch(params, ns, pts):
    """
    Instantaneous population size change, plus selection.

    params: [nu,T,gamma]
    ns: Sample sizes
    pts: Grid point settings for integration

    Note that DFE methods internally apply make_extrap_func,
    So there is no need to make it extrapolate again.

    nu: Final population size
    T: Time of size change
    """
    nu, T, gamma = params
    xx = Numerics.default_grid(pts)
    phi = PhiManip.phi_1D(xx, gamma=gamma)
    phi = Integration.one_pop(phi, xx, T, nu, gamma=gamma)
    fs = Spectrum.from_phi(phi, ns, (xx,))
```

Then, we generate the frequency spectra for a range of gammas. The following code generates expected frequency spectra, conditional on the demographic model fit to synonymous sites, over `gamma_pts` log-spaced points over the range of `gamma_bounds`. Additionally, the `mp=True` argument tells fit∂a∂i whether it should utilize multiple cores/threads, which is convenient since this step takes the longest. If the argument `cpus` is passed, it will utilize that many cores, but if `mp=True` and no `cpus` argument is passed, it will use `n-1` threads, where `n` is the number of threads available. If `mp=False`, each SFS will be computed in serial. This step should take 1-10 minutes depending on your CPU.

```
    ns = [250]
```

```
    # Integrate over a range of gammas
    pts_l = [600, 800, 1000]
```

Note, one error message that will come up often with very negative selection coefficients is:

```
WARNING:Numerics:Extrapolation may have failed.  Check resulting frequency spectrum
for unexpected results.
```

One way to fix this is by increasing the `pts_l` grid sizes – this will need to increase as the sample size increases and/or if the integration is done over a range which includes stronger selection coefficients. `dadi.Numerics.make_extrap_func` is used to extrapolate the frequency spectra to infinitely many gridpoints, but will sometimes return tiny negative values (often $|X_i| < 1e{-}50$) due to floating point rounding errors. Using `dadi.Numerics.make_extrap_log_func` will sometimes return `Inf` values and is harder to work with. In practice, it seems that the tiny negative values do not affect the integration because they are insignificant, but if the error message appears it is good to manually double-check each SFS. Alternately, the small negative values can be manually approximated with 0.

In the example, the pre-computed SFSs are saved in the list `spectra.spectra`. For convenience's sake, the `spectra` object can be pickled.

```
                                mp=True)
    # The spectra can be pickled for usage later. This is especially conv
```

12.4 Fitting a DFE

12.4.1 Fitting simple DFEs

Fitting a DFE is the quickest part of this procedure, especially for simple distributions such as the gamma distribution. If you wish to get an SFS for a specific DFE, you can use the `integrate` method that is built into the spectra object: `spectra.integrate(sel_params, None, sel_dist, theta, None)`. `sel_params` is a list containing the DFE parameters, `sel_dist` is the distribution used for the DFE, and `theta` is θ_{NS}. To compute the expected SFS for our simulations with the true parameter values, we would use `spectra.integrate([0.186, 686.7], None, Selection.gamma_dist, 4000., None)`. (The `None` arguments are for `ns` and `pts`, which are ignored. These are useful to ensure compatibility with dadi's optimization functions.)

First, load the sample data:

Similar to the way in which vanilla ∂a∂i is used, you should have a starting guess at the parameters. Set an upper and lower bound. Perturb the parameters to select a random starting point, then fit the DFE. This should be done multiple times from different starting points. We use the `spectra.integrate` methods to generate the expected SFSs during each step of the optimization. The following lines of code fit a gamma DFE to the example data:

```
    # Fit a DFE to the data
    # Initial guess and bounds
```

```
sel_params = [0.2, 1000.]
lower_bound, upper_bound = [1e-3, 1e-2], [1, 50000.]
p0 = dadi.Misc.perturb_params(sel_params, lower_bound=lower_bound,
                              upper_bound=upper_bound)
popt = dadi.Inference.optimize_log(p0, data, spectra.integrate, pts=N
```

If this runs correctly, you should infer something close to, but not exactly, the true DFE. The final results will be stored in `popt`. The expected SFS at the MLE can be computed with:

```
                                          verbose=len(sel_params), maxiter=1
```

12.4.2 Fitting complex DFEs

Fitting complex distributions is similar to fitting simple DFEs, but requires a few additional steps. The following code can be used to fit a neutral+gamma mixture DFE to the data. Note that the gamma DFE should fit better if assessing model fit using AIC. Additionally, we assume that every selection coefficient $\gamma < 1e-4$ is effectively neutral. Since this is a mixture of two distributions, we infer the proportion of neutral mutations, p_{neu}, and assume the complement of that (i.e. $1 - p_{neu}$) is the proportion of new mutations drawn from a gamma distribution. Therefore, the parameter vector should be: [p_{neu},shape, scale]. The gamma DFE is still the true DFE.

```
# One possible characterization of the neutral+gamma DFE
# Written using numpy tricks to work with both scalar and array argum
def neugamma(xx, params):
    pneu, alpha, beta = params
    # Convert xx to an array
    xx = np.atleast_1d(xx)
    out = (1-pneu)*DFE.PDFs.gamma(xx, (alpha, beta))
    # Assume gamma < 1e-4 is essentially neutral
```

Fit the DFE as before, accounting for the extra parameter to describe the proportion of neutral new mutations. Note that p_{neu} is explicitly bounded to be $0 < p_{neu} \leq 1$.

```
    # Reduce xx back to scalar if it's possible
    return np.squeeze(out)

sel_params = [0.2, 0.2, 1000.]
lower_bound, upper_bound = [1e-3, 1e-3, 1e-2], [1, 1, 50000.]
p0 = dadi.Misc.perturb_params(sel_params, lower_bound=lower_bound,
                              upper_bound=upper_bound)
popt = dadi.Inference.optimize_log(p0, data, spectra.integrate, pts=N
                                   func_args=[neugamma, theta_ns],
```

For fitting with ancestral state misidentification or including a point mass of positive selection, see `example1D.py`.

12.5 Fitting joint DFEs

dadi can also fit joint DFEs between populations, in a similar fashion to one-dimensional DFEs.

Caching is similar to the one-dimensional case, although note that it is generally much more computationally expensive.

```
try:
    s2 = pickle.load(open('test.spectra2d.bpkl', 'rb'))
except IOError:
    s2 = Cache2D(demo_params, ns, func_ex, pts=pts_l, gamma_pts=100,
                 gamma_bounds=(1e-2, 10), verbose=True, mp=True,
                 additional_gammas=[1.2, 4.3])
    # Save spectra2d object
```

dadi currently includes a few simple models for joint DFEs in `dadi.DFE.PDFs`. Note that the semi-analytic integration of the distribution over the regime not covered by the cache is expensive. Therefore, C implementations of the PDFs can make a big difference in computational time, and we provide C implementations for the default PDFs.

Calculating individual spectra is very similar to the 1D case.

```
p0 = [0,1.,0.8]
popt = dadi.Inference.optimize(p0, data, s2.integrate, pts=None,
                               func_args=[sel_dist, theta],
```

As is optimization.

```
#
# Test optimization of point mass positive selection.
#

# Generate test data set to fit
# This is a symmetric case, with mu1=mu2=0.5, sigma1=sigma2=0.3, rho=
# ppos1=ppos2=0.2, gammapos1=gammapos2=1.2.
input_params, theta = [0.5,0.3,-0.5,0.2,1.2], 1e5
# Expected sfs
target = s2.integrate_symmetric_point_pos(input_params, None, sel_dist
                                          pts=None)
# Get data with Poisson variance around expectation
data = target.sample()
```

Note that when a point mass of positive selection is included in a 2D DFE, the assumed value for the positive γ must be cached, otherwise evaluation would be too expensive.

dadi also implements mixture models, in which the total DFE is a sum of a 2D distribution plus a 1D distribution representing perfect correlation. These are implemented by `dadi.DFE.mixture`.

```
                               lower_bound=[-1,0.1,-1,0,0],
                               upper_bound=[1,1,1,1,None],
```

Using a mixture model requires both a 1D and a 2D cache.

For additional examples, see `examples/DFE/example2D.py` in the source distribution.

13 Inbreeding

When populations are inbred, the excess homozygosity can distort the SFS such that the even entries are greater than the odd entries for a population. For high levels of inbreeding ($F_{IS} \approx 0.5$ or higher), this will generate frequency spectra with a conspicuous pattern of zig-zagging up and down between adjacent entries. However, lower levels of inbreeding can still bias estimates of demography despite not have such a dramatic effect on the SFS.

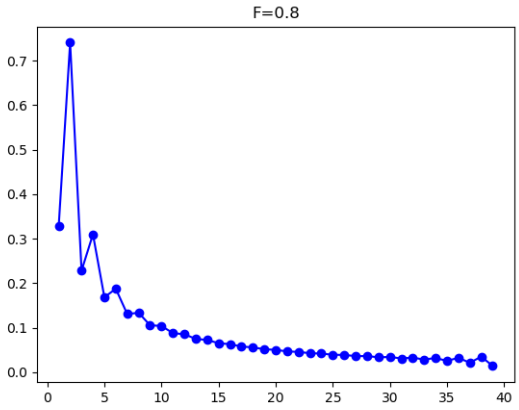

Figure 7: **SFS with Inbreeding:** $F_{IS} = 0.8$.

To accommodate this, inbreeding can be estimated as part of a demographic model by using the `from_phi_inbreeding` function in the `Spectrum` class. This can be done by including additional parameters for the inbreeding coefficients (one for each population) in the model and passing them to the `from_phi_inbreeding` function in the demographic model:

```
def snm_inbreeding(params, ns, pts):
    F = params[0]
    xx = dadi.Numerics.default_grid(pts)
    phi = dadi.PhiManip.phi_1D(xx)
    fs = dadi.Spectrum.from_phi_inbreeding(phi, ns, (xx,),
                                           (F,), (2,))
    return fs
```

Listing 9: **Inbreeding:** Standard neutral model for a diploid population with inbreeding level `F`.

The `from_phi_inbreeding` function also requires the ploidy levels of the populations being analyzed, so it can naturally handle autopolyploids as well:

```
def iso_inbreeding(params, ns, pts):
    T, nuA, nuB, F1, F2 = params
    xx = dadi.Numerics.default_grid(pts)
    phi = dadi.PhiManip.phi_1D(xx)
    phi = dadi.PhiManip.phi_1D_to_2D(xx, phi)
    phi = phi.Integration.two_pops(phi, xx, T, nu1, nu2)
    fs = dadi.Spectrum.from_phi_inbreeding(phi, ns, (xx,xx),
                                           (F1,F2), (2,4))
    return fs
```

Listing 10: **Diploid-Tetraploid Isolation Model:** An ancestral population splits at time `T` into a diploid (pop 1) and autotetraploid (pop 2) population of sizes `nu1` and `nu2`, respectively. The populations have separate inbreeding coefficients `F1` and `F2`.

14 Polyploid Subgenomes

To model polyploid lineages, a demographic model for their subgenomes can be created. This is done by treating the subgenomes as analogous to populations, and then combining their frequency spectra into a single, polyploid SFS. Including a migration parameter between the subgenomes can act as a proxy for allelic exchange (homoeologous recombination), which allows both autopolyploids and allopolyploids to be modeled. The input for a single polyploid lineage is a one-dimensional SFS, so there is no need to predetermine if the lineage is auto- or allopolploid. There is also no need to try and separate SNP calls between subgenomes because fixed heterozygosity is naturally accommodated by this model.

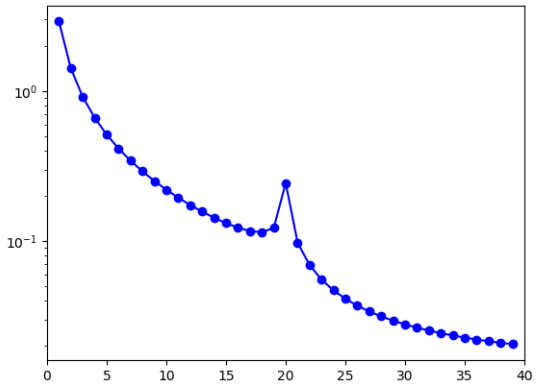

Figure 8: **Tetraploid SFS:** $F_{IS} = 0.8$.

```
def two_subgenomes(params, ns, pts):
    T, m = params
    xx = dadi.Numerics.default_grid(pts)
```

```
    phi = dadi.PhiManip.phi_1D(xx)
    phi = dadi.PhiManip.phi_1D_to_2D(xx, phi)
    phi = dadi.Integration.two_pops(phi, xx, T, 1.0, 1.0, m, m)
    fs = dadi.Spectrum.from_phi(phi, ns, (xx,xx)
    fs2 = dadi.Spectrum(dadi.Misc.combine_pops(fs))
    return fs2
```

Listing 11: **Two subgenomes:** At time `T` in the past, an equilibrium population duplicates (autopolyploidy) and the subgenomes exchange genes symmetrically at a rate of `m`. The SFS for the subgenomes are then combined with the `combine_pops` function to create a single, polyploid SFS.

15 Installation

15.1 Dependencies

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

- Python 3
- NumPy
- SciPy

It is also recommended that you install

- matplotlib
- IPython

The easiest way to obtain all these dependencies is to install the Anacond Python Distribution. The easiest way to install dadi is then via `conda`, using the command `conda install -c bioconda dadi`. dadi can also be installed via `pip`, using the command `python3 -m pip install dadi`.

15.2 Installing from source

dadi can be easily installed from source code, as long as you have an appropriate C compiler installed. (On OS X, you'll need to install the Developer Tools to get `gcc`.) To do so, first unpack the source code tarball, `unzip dadi-<version>.zip`. In the `dadi-<version>` directory, run `sudo python setup.py install`. This will compile the C modules dadi uses and install those plus all dadi 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`

16 Frequently asked questions

1. What does the message `WARNING:Inference:Model is < 0 where data is not masked.` mean?

   This warning comes from the likelihood calculation function. It indicates that the model frequency spectrum has negative values that are trying to be compared with data. Negative values in the frequency spectrum are nonsense, so this most likely indicates numerical difficulties. If you're running an optimization, occasional warnings like this likely not a problem. The optimization explores a wide range of parameter values, most of which are bad fits. If these errors crop up for parameter values that will be a bad fit anyways, the errors won't change the final result. On the other hand, if you are re getting these warnings near good-fitting sets of parameters, you'll need to fix them. There are two possible causes, and thus two possible solutions.

   (a) The negative values might be arising from the extrapolation process (over different grid sizes). In this case, replace any calls to `Numerics.make_extrap_func` to `Numerics.make_extrap_log_func`. This will do the extrapolation based on the logarithms of the value in the frequency spectrum, guaranteeing positive results.

   (b) The negative values might be arising from calculating an individual spectrum (for a fixed grid size). This typically only happens in cases of very rapid exponential growth. In this case, you can try a finer grid size (increase the elements of the `pts_l` list) or smaller time steps. The time step is set by the call to `dadi.Integration.set_timescale_factor(pts_l[-1], factor=10)`. To shorten the time step, increase `factor`. First try shortening the time step, as this will typically increase computation time less than increasing the grid size.

2. I'm projecting my data down to a smaller frequency spectrum. What sample sizes should I project down to?

   At this time, we have not done any formal power testing to judge the optimal level of projection, but we do have a rule-of-thumb. As you project down to smaller sample sizes, more SNPs can be used in constructing the FS (because they have enough successful calls). However, as you project downward, some SNPs will "disappear" from the FS because they get partially projected down to 0 observations in all the populations. Our rule of thumb is to use the projection that maximizes the number of segregating SNPs. The number of segregating SNPs can be calculated as `fs.S()`.

References

[1] Hernandez RD, Williamson SH, Bustamante CD (2007) Context dependence, ancestral misidentification, and spurious signatures of natural selection. Mol Biol Evol 24: 1792–1800.

[2] Weir BS, Cockerham CC (1984) Estimating F-statistics for the analysis of population structure. Evolution 38: 1358–1370.

[3] Gutenkunst RN, Hernandez RD, Williams SH, Bustamante CD (2009) Inferring the joint demographic history of multiple populations from multidimensional SNP frequency data. PLoS Genet 5: e1000695.

[4] Pierce DA, Schafer DW (1986) Residuals in generalized linear models. J Am Stat Assoc 81: 977–986.

[5] Ku HH (1966) Notes on the use of propagation of error formulas. J Res Nbs C Eng Inst 70: 263-273.

[6] Coffman AJ, Hsieh P, Gravel S, Gutenkunst RN (2016) Computationally efficient composite likelihood statistics for demographic inference. Mol Biol Evol 33: 591–593.

[7] Self SG, Liang Ky (1987) Asymptotic properties of maximum likelihood estimators and likelihood ratio tests under nonstandard conditions. J Am Stat Assoc 82: 605–610.

[8] Kim BY, Huber CD, Lohmueller KE (2017) Inference of the distribution of selection coefficients for new nonsynonymous mutations using large samples. Genetics 206: 345.

[9] Ortega-Del Vecchyo D, Marsden CD, Lohmueller KE (2016) Prefersim: Fast simulation of demography and selection under the poisson random field model. Bioinformatics : btw478.

[10] Haller BC, Messer PW (2016) Slim 2: Flexible, interactive forward genetic simulations. Molecular Biology and Evolution : msw211.