Utilizing fit∂a∂i: a short how-to guide

Bernard Kim
bernard.kim@ucla.edu

Updated: November 8, 2018

1 Introduction

This short guide outlines how to utilize the code we have made available for inferring of the distribution of fitness effects (DFE). This code serves as an add-on module for the package ∂a∂i [1], which is primarily used for demographic inference.

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 [2]. 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 [3].

If you have any additional questions, please feel free to email us: Bernard Kim (bernard.kim@ucla.edu) or Kirk Lohmueller (klohmueller@ucla.edu).

2 Installation

∂a∂i can be downloaded from the Gutenkunst Lab's BitBucket site. Once you have ∂a∂i, there are two easy ways to utilize the fit∂a∂i module. The first method allows you to set up fit∂a∂i so that it is installed with the ∂a∂i code. To do this, you copy `Selection.py` and `__init__.py` into the `dadi` directory before installing ∂a∂i with `python setup.py install`. Note, it may be useful to comment out the line `import matplotlib` in `__init__.py` if you are trying to use ∂a∂i on a cluster. Alternately, you can import our code separately. To do this, you should install ∂a∂i in the standard manner. Copy `Selection.py` into your working directory. Then, our module can be loaded by including the line `import Selection.py` into your ∂a∂i/Python script.

The latter method is the way in which the example script is set up:

```
1 import numpy
2 import dadi
3 import Selection
```

Note that in order to run ∂a∂i with the example script you must also copy `Selection.py` into the same directory as the example script.

3 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 paramter **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 $\theta_{NS} = 4000 = 4N_A\mu L_{NS}$.

4 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. Although we are skipping the demographic inference, the following ∂a∂i function describes a two epoch demographic model.

```
def two_epoch(params, ns, pts):
    nu, T = params
    xx = dadi.Numerics.default_grid(pts)
    phi = dadi.PhiManip.phi_1D(xx)
    phi = dadi.Integration.one_pop(phi, xx, T, nu)
    fs = dadi.Spectrum.from_phi(phi, ns, (xx,))
    return fs
```

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]`.

```
demog_params = [2, 0.05]
```

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).

```
theta_ns = 4000.
ns = numpy.array([250])
```

5 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 s is constant, the same `gamma` should be used throughout the demographic function. If `gamma=0`, this function is the same as the original demographic function.

```
def two_epoch_sel(params, ns, pts):
    nu, T, gamma = params
    xx = dadi.Numerics.default_grid(pts)
    phi = dadi.PhiManip.phi_1D(xx, gamma=gamma)
    phi = dadi.Integration.one_pop(phi, xx, T, nu, gamma=gamma)
    fs = dadi.Spectrum.from_phi(phi, ns, (xx,))
    return fs
```

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 `Npts` log-spaced points over the range of `int_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.

```
pts_l = [600, 800, 1000]
spectra = Selection.spectra(demog_params, ns, two_epoch_sel, pts_l=pts_l,
                            int_bounds=(1e-5,500), Npts=300, echo=True,
                            mp=True)
```

Another manner in which this can be done is multiple log-spaced grids over some predetermined breaks. While this method is almost the same as the previous, this method is more appropriate for discretely binned DFEs. For example, if we were to create discrete bins at the breaks $\gamma = [0.1, 1, 100]$, we would use the following command, importantly, passing a specific `int_breaks`.

```
pts_l = [600, 800, 1000]
int_breaks = [1e-4, 0.1, 1, 100, 500]
spectra = Selection.spectra(demog_params, ns, two_epoch_sel, pts_l=pts_l,
                            int_breaks=int_breaks, Npts=300,
                            echo=True, mp=True)
```

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 care should be taken that these numerical errors do not propagate to downstream steps of the analysis. In practice, it seems that the tiny negative values do not affect the integration because they are insignificant, but if the error message appears the SFS should be manually double-checked. Alternately, the small negative values can be manually replaced 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.

```
#import pickle
#pickle.dump(spectra, open("example_spectra.sp","w"))
```

6 Fitting a DFE

6.1 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, sel_dist, theta)`. Another option is to use the `spectra.integrate_norm` method. The former does not normalize the DFE and the second normalizes the DFE. We chose to use the former and assumed that mutations with selection coefficients outside of the range we integrated over were effectively lethal, that is, not seen in the sample. Note that we integrated over the range of gammas corresponding to $|s| = [0, 1]$. `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], Selection.gamma_dist, 4000.)`.

First, load the sample data:

```
data = dadi.Spectrum.from_file('example.sfs')
```

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:

```
sel_params = [0.2, 1000.]
lower_bound = [1e-3, 1e-2]
upper_bound = [1, 50000.]
p0 = dadi.Misc.perturb_params(sel_params, lower_bound=lower_bound,
                              upper_bound=upper_bound)
popt = Selection.optimize_log(p0, data, spectra.integrate, Selection.gamma_dist,
                              theta_ns, lower_bound=lower_bound,
                              upper_bound=upper_bound, verbose=len(sel_params),
                              maxiter=30)
```

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

```
>>> popt
[-678.338 , array([ 0.187071 , 666.092 ])]
```

The expected SFS at the MLE can be computed with:

```
model_sfs = spectra.integrate(popt[1], Selection.gamma_dist, theta_ns)
```

6.2 Fitting complex DFEs

Fitting complex distributions is similar to fitting simple DFEs, but requires a few additional steps, outlined in the following lines of code. Here, we are fitting a neutral+gamma DFE to an SFS generated under a gamma DFE just for the sake of the example. 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 is: [p_{neu},shape, scale].

```
def neugamma(mgamma, pneu, alpha, beta):
    mgamma=-mgamma
    #assume anything with gamma<1e-4 is neutral
    if (0 <= mgamma) and (mgamma < 1e-4):
        return pneu/(1e-4) + (1-pneu)*Selection.gamma_dist(-mgamma, alpha,
                                                           beta)
    else:
        return Selection.gamma_dist(-mgamma, alpha, beta) * (1-pneu)
```

Then, the custom DFE needs to be vectorized. This is easily done with the `numpy.frompyfunc` function.

```
neugamma_vec = numpy.frompyfunc(neugamma, 4, 1)
```

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 $0 < p_{neu} \leq 1$.

```
sel_params = [0.2, 0.2, 1000.]
lower_bound = [1e-3, 1e-3, 1e-2]
upper_bound = [1, 1, 50000.]
p0 = dadi.Misc.perturb_params(sel_params, lower_bound=lower_bound,
                              upper_bound=upper_bound)
popt = Selection.optimize_log(p0, data, spectra.integrate, neugamma_vec,
                              theta_ns, lower_bound=lower_bound,
                              upper_bound=upper_bound, verbose=len(sel_params),
                              maxiter=30)
```

If this has run properly, your result should look like the following:

```
>>> popt
[-678.35389753002551, array([ 1.10498940e-03, 1.87041580e-01, 6.71354664e+02])]
```

Another way to approach this problem is by defining the neutral+gamma DFE as a function of four parameters instead of assuming the two are complementary. While this is unnecessary for the neutral+gamma distribution since we bounded p_{neu} to be between 0 and 1, it becomes necessary for mixtures of three or more distributions. However, the principles will be the same as shown here.

First, define the neutral+gamma DFE as a function with parameter vector: [p_{neu},p_{gamma},shape, scale].

```
def neugamma(mgamma, pneu, pgamma, alpha, beta):
    mgamma=-mgamma
    #assume anything with gamma<1e-4 is neutral
    if (0 <= mgamma) and (mgamma < 1e-4):
        return pneu/(1e-4) + pgamma*Selection.gamma_dist(-mgamma, alpha, beta)
    else:
        return Selection.gamma_dist(-mgamma, alpha, beta) * pgamma
```

Here we do not explicitly set $p_{neu}+p_{gamma}=1$, so we need to apply some additional constraints to enforce this. A function that equals 0 when the constraint is satisfied should be used:

```
def consfunc(x, *args):
    return 1-sum(x[0:-2])
```

In other words, the constraint is satisfied when `sum([`p_{neu}`,`p_{gamma}`,shape, scale][0:-2])`=1, that is, when `sum([`p_{neu}`,`p_{gamma}`])`=1.

Then, use the function `Selection.optimize_cons` to fit the DFE. This function is the same as `Selection.optimize` except that it requires the constraint function to be passed as an additional argument.

```
neugamma_vec = numpy.frompyfunc(neugamma, 5, 1)

sel_params = [0.2, 0.8, 0.2, 1000.]
lower_bound = [1e-3, 1e-3, 1e-3, 1e-2]
upper_bound = [1, 1, 1, 50000.]

p0 = dadi.Misc.perturb_params(sel_params, lower_bound=lower_bound,
                              upper_bound=upper_bound)
popt = Selection.optimize_cons(p0, data, spectra.integrate, neugamma_vec,
                               theta_ns, lower_bound=lower_bound,
                               upper_bound=upper_bound, verbose=len(sel_params),
                               maxiter=30, constraint=consfunc)
```

The result is roughly similar to the previous method:

```
>>> popt
[-678.40252289901571, array([ 1.00000000e-03, 9.99000000e-01, 1.85534565e-01,
6.93897711e+02])]
```

References

[1] Ryan N Gutenkunst, Ryan D Hernandez, Scott H Williamson, and Carlos D Bustamante. Inferring the joint demographic history of multiple populations from multidimensional snp frequency data. *PLoS Genet*, 5(10):e1000695, 2009.

[2] Diego Ortega-Del Vecchyo, Clare D Marsden, and Kirk E Lohmueller. Prefersim: Fast simulation of demography and selection under the poisson random field model. *Bioinformatics*, page btw478, 2016.

[3] Benjamin C Haller and Philipp W Messer. Slim 2: Flexible, interactive forward genetic simulations. *Molecular Biology and Evolution*, page msw211, 2016.