Covariance and Sampling Code

November 8, 2018

1 Introduction

The sampling code is divided into two classes, `Covariance` (Covariance.py), and `Sampler`, (Sampler.py). The covariance object loads a power spectrum from a file, and computes value- and derivative- biased covariance matrices for the radial mode coefficients of the associated gaussian random field. These matrices are computed mode-by-mode (in the spherical harmonic decomposition), over a given radial grid. The same object then computes square-roots of these matrices, and can save and load the results of these computations to a .npz file. `Covariance` is intended to be used to perform these computations, and then also to load and manage the resulting files.

The second class, `Sampler`, instantiates `Covariance`, and uses it to access the roots of the covariance matrices. Sampler then generates random samples of the gaussian modes given a field height bias $\bar{\nu}u$ and field number N. These modes are squared and summed pointwise to yield discrete samples of $\Phi_{00}(r)$, over a the radial grid. Note that field height $\bar{\nu}u$ and field number N may be altered in the calls to `Sampler`, without needing to recompute the covariance matrices. On the other hand, any changes to the power spectrum, its frequency domain, or the radial grid, will require recomputation of the covariance matrices and their roots.

This document will outline the structures of these classes (as well as briefly discussing the classes which perform Levin integration), provide some comments and notes about their design, and specify issues that need to be resolved (minor issues, many of which have to do with interface). At the end, we'll present a comparison between the sample statistics generated by the code and the analytic statistics from the paper, and argue that these classes are behaving as expected.

2 Covariance

2.1 Functions

__**init()**__ The initializer doesn't take any arguments, but is rather used as a place to hard-code some parameters. In particular, the frequency-space resolution `spectrum_spline_resolution` used to spline-fit the power spectrum, the factor by which to increase grid resolution when spline-fitting the correlations $\rho_C(r)$ and $\rho_D(r)$. **FIXME: should we also define a precision target here, or pass it to** `ComputeCovariances()`?

ComputeCovariances(fname_spectrum, k_low, k_high, l_max, r_max, r_step)
This is the "main" function of the class. It calls all the other functions required to compute the covariance data in order. Argument `fname\_spectrum` is a filename for the power spectrum data, `k\_low` is the lower frequency cutoff to use when doing computations involving the power spectrum, `k\_high` is the corresponding upper cutoff frequency, `l\_max` is the maximum spherical harmonic ℓ mode to include, `r\_max` is the maximum radius in the grid (minimum is always at zero), and `r\_step` is the spacing between grid points. Note that both endpoints are included, so the number of gridpoints will be . The computation then takes place in steps,

1. Compute moments of the power spectrum
2. Initialize the radial grid from `r_max` and `r_step`
3. Fit splines to ρ_C and ρ_D over the appropriate radial domains with calls to `ComputeRhoCSpline()` and `ComputeRhoDSpline()`.
4. Compute biased covariance matrices for each ℓ mode up to `l_max` with call to `ComputeCovarianceMatrices()`
5. Compute a square root of the covariance matrix: a matrix A such that $AA^T = \Sigma$, with call to `ComputeSqrtCovariances()`. The covariance matrices and their square roots are stored as object member variables (See `ComputeCovarianceMatrices()` and `ComputeSqrtCovariances()`).

LoadDummyPowerSpectrum() FIXME: This is a testing function, to be replaced by LoadPowerSpectrum(). Currently, this function fits a spline to the function $\exp(-k)$ over the region [`k_low`, `k_high`]. The fit is done with resolution `self.spectrum_spline_resolution`, which is set in the constructor. Spectrum loaded to variable `self.spectrum`.

LoadPowerSpectrum() FIXME: write this! This will load the power spectrum from the file specified by `self.fname_spectrum`, which is passed to `ComputeCovariances()`. A spline will then be fit between `k_low` and `k_high` with `self.spectrum_spline_resolution` points. **FIXME: maybe use k_step instead, and pass as an argument to ComputeCovariances()?** I'm waiting to see whether we use $k, \mathcal{P}(k)$ pairs, or $k, k^2\mathcal{P}(k)$ pairs to write this function, and to conform to the final choice of format for the spectrum file. Spectrum will be loaded to variable `self.spectrum`.

ComputeFrequencyCutoff(power_fraction) This function computes an upper cutoff frequency k_{cut} based on the loaded power spectrum spline and a fraction $\alpha =$ `power_fraction` such that

$$\int_{k_{\text{bottom}}}^{k_{\text{cut}}} \mathcal{P}(k)\, dk = \alpha. \tag{1}$$

This function is only here for convenience, and I'm guessing we may want to move its functionality earlier in the stack, so that the upper and lower bound frequencies in the spectrum file itself are already determined according to some reasonable physical criteria (like this one)

> **FIXME: note, this integral has as lower bound the lowest k available in the spline, not k_low as passed to ComputeCovariances().** Returns k_{cut}.

ComputeSigmaN(n) This function computes the nth moment of the power spectrum according to Equation (7) in arXiv:1810.02078. Returns the moment.

ComputeRhoC(r) Computes $\rho_C(r)$ as defined in Equation (82) of arXiv:1810.02078. Uses the Levin integration method `ICalc` defined in the `levin` module. Returns $\rho_C(r)$.

ComputeRhoD(r) Computes $\rho_D(r)$ as defined in Equation (82) of arXiv:1810.02078. Uses the Levin integration method `ICalc` defined in the `levin` module. Returns $\rho_D(r)$.

ComputeRhoCSpline() First computes a high-resolution discretization of ρ_C over the range $[0, 2r_{\text{max}}]$. The resolution is increased from the r_{step} resolution by a factor of `self.hires_factor`, which is set in the constructor. Fits a spline approximating ρ_C to this discretization. The computation of the discretization is conditional on the corresponding variable having NoneType. Since this variable is saved in `Save()`, this saves time if the array has been loaded. We save this array in addition to the covariance data because it is useful for computing analytics later on (for comparison to the samples). Spline stored at `self.rho_c_spline`.

ComputeRhoDSpline() Same as `ComputeRhoCSpline()`, but for ρ_D, as defined in Equation (82) of arXiv:1810.02078, and over the range $[0, r_{\text{max}}]$. Spline stored at `self.rho_d_spline`.

ComputeRhoCLElement Computes $\rho_{C_\ell}(r, r') = \tilde{C}_\ell(r, r')/\sigma_0^2$, where $\tilde{C}_\ell$ is given in Equation (138) of arXiv:1810.02078. The integral in this definition is not badly oscillatory, and is done using the SciPy quad() routine. The integrand function evaluates the $\rho_C(r)$ spline, which is why we pre-compute it! Returns $\rho_{C_\ell}(r, r')$.

ComputeCovarianceMatrices() Computes biased covariance matrix elements according to Equation (165) of arXiv:1810.02078. Calls `ComputeRhoCLElement()` and evaluates the ρ_C and ρ_D splines in the process. The elements are stored in two NumPy different arrays, `self.Covariances_leq1` and `self.Covariances_lneq1` depending on whether or not $\ell = 1$ (which changes the biasing conditions). Each array has three indices. The first indicates which α field space component is being considered, which also alters the biasing conditions. The second and third indices are the radial grid indicies corresponding to the given covariance element.

ScrubNegativeEigenvalues(eigvals) Takes a list of eigenvalues, `eigvals`, and checks that all negative eigenvalues have absolute value $|\lambda| < 1e - 6$. Otherwise, throws an assertion. **FIXME: shouldn't hardcode this value. We may want to feed some kind of precision target to the Covariance object and base it off of that value.** If the negative eigenvalues are below this absolute value threshold, they are

replaced by 0.0. This compensates for negative eigenvalues arising from small numerical errors in the diagonalization of the covariance matrices performed in `ComputeSqrtMatrix()`. Returns an array of scrubbed eigenvalues, `clean_eigvals`.

ComputeSqrtMatrix(mat) takes a 2D NumPy array `mat` and computes matrices M and Λ (lambda diagonal) such that $M\Lambda M^T = $ mat. Scrubs negative eigenvalues from Λ, and computes $A = M\sqrt{\Lambda}$, which satisfies $AA^T = $ mat. Used by `ComputeSqrtCovariances()`. Returns A.

ComputeSqrtCovariances() Computes the A matrices corresponding to each of the covariance matrices, making calls to `ComputeSqrtMatrix()`. Stores the A matrices in the same format as the covariance matrices, in variables `self.sqrtcov_lneq1` and `self.sqrtcov_leq1`.

GetSqrtCovariance(l,alpha) A convenience function to simplify access to the A matrices. Essentially just conditionals that take a pair ℓ, α and return the corresponding biased covariance matrix.

Save(fname) Saves results from the calculations detailed above as NumPy arrays in a .npz file with filename specified by `fname`.

Load(fname) Loads results from the calculations detailed above from a NumPy array in a .npz file specified by `fname`. Results are loaded to their corresponding object member variables, so if we pre-compute covariance data, then call `Save()`, and then instantiate a new `Covariance` object and call `Load()`, the results of the computation can be accessed in the same way as in the original object. Some caveats: splines can't be saved to .npz files, and so must be re-computed from the discretizations. Also, the spectrum must be re-loaded if access to the spectrum is desired. **Note: for "production" usage, we will mainly just need to retrieve the A matrices from the Covariance object, so we can just call Load() and then GetSqrtCovariance()**

3   Levin

I wont say too much about the Levin integration here (Levin.py), except to highlight the most important functions. There are two classes, `LevinIntegrals`, which actually implements the levin method and the attendant collocation, and `SphericalBesselIntegrator`, which implements an adaptive integration of "I-type" integrals. That is,

$$I(\alpha) = \int_a^b dk \, f(k) j_\ell(\alpha * k). \tag{2}$$

3.1   LevinIntegrals

__init__(numpoints) Initialize variables for the computation. Most importantly, takes [numpoints] which is a positive integer determining the number of points to use in the collocation grid.

4

compute_I(a,b,alpha,l,func) Performs a levin integration of the I type integral determined by limits [a,b], "radial parameter" [alpha] (this will be the radius when we compute $\rho_C(r), \rho_D(r)$), spherical harmonic mode [l], and function [func] (which will be the power spectrum for our purposes). Returns the result of the integral.

3.2 SphericalBesselIntegrator

This class wraps the `LevinIntegrals` function `compute_I()` and provides case-checking and adaptive collocation resolution.

__init__() sets initial collocation resolution, as well as maximum recursion depth. Instantiates a LevinIntegrals object at the collocation resolution corresponding to each recursion level. Also sets relative error tolerance. **FIXME: the relative tolerance should depend on some precision target we pass to our code at the beginning of our computations. We need to propagate this change here and elsewhere in the stack.**

ICalc(a,b,alpha,l,func) Takes integration limits [a,b], radial coefficient [alpha], mode [l], and function [func] and computes the integral specified in Equation (2). First, we check whether $\alpha == 0.0$ and $\ell = 0$. If so, the integral reduces to an integral over the power spectrum (func), and we switch to quadrature **FIXME: I think this was done because the levin method was not as accurate as quadrature here, but I'm not sure how much worse it is/ if this conditional is really needed.** Otherwise, we begin with the two lowest-resolution LevinIntegrals objects, and perform the integral. If the difference is below tolerance, we return the higher precision result. Otherwise, we recurse and take the next lowest resolution pair, and compare the difference to tolerance, and so on. Either we eventually suppress estimated error below tolerance, or hit the recursion limit. If the former, we return the result. If the latter, we make a last-ditch attempt with quadrature, and if that fails, throw an assertion. **Note: The levin method fails when either r or k is zero (r*k appears in the argument to the spherical bessel function). The r=0 case is handled in the levin integrator code, but we get division by zero when the lower bound for k is zero. I think this never actually happens for real power spectra (the lowest k we feed into the covariance code is not zero), but we need to be aware of this!**

Note: There were originally two special cases in the levin code. First, $\alpha = 0$ and $\ell = 0$. This case reduces to an integral over the power spectrum, which is handled just fine. The second case (since removed) had some cutoff for b, the upper limit of momentum space. If b were less than this cutoff, the code would resort to quadrature. I vaguely remember the levin method losing out to quadrature in the low-k regime (where the integrand does not oscillate), but I worried that this cutoff was arbitrary. *update*: I did some preliminary checking against mathematica, and it doesn't seem to be the

case that this outperforms levin in the stated regime (with an exponential dummy spectrum). So, I got rid of this second conditional. Note that the levin method will resort to quadrature anyway if the maximum number of collocation recursions is exceeded, so if the b limit becomes a problem, and quadrature really is the way to go, the code should switch automatically.

Note: The levin integration can take a long time for large α if we use a low resolution spline to fit the power spectrum (this showed up in testing against the exponential dummy spectrum). The moral of the story is that we should try upping spectrum resolution if the integrals are hanging.

4 Sampler

This class is short and sweet. It simply extracts precomputed data from a Covariance object, carries out the gaussian sampling, and the subsequent construction of the $\Phi_{00}(r)$ sample.

__init__(covariance) Extracts some data from the Covariance object [covariance] and stores a copy as `self.covariance`.

GetSamples(nubar,nfields,nsamples) Takes a dimensionless field height [nubar], number of waterfall fields [nfields], and number of samples [nsamples], and returns a 2D array of samples of Phi_{00} over the radial grid specified in the Covariance object. The first index points to the sample number, and the second is the radial index within a given sample.

5 Testing

Ok, so I finally got around to doing a serious check of the covariance and sampling code! I began with an exponential dummy spectrum,

$$\mathcal{P}(k) = e^{-k}, \tag{3}$$

where the lower and upper cutoffs for k are given by $k_{lo} = 0.001$, $k_{hi} = 5.0$. Note: I need to implement the loading of the actual power spectrum. Will this be $\mathcal{P}(l)$ or $k^2\mathcal{P}(k)$? I'll need to alter the calculations accordingly. First, I computed the mean of 1000 samples and compared the mean as a function of radius (together with standard error) to the analytic result for truncated ℓ given in Equation (184) of arXiv:1810.02078. The comparison is shown in Figure 1. The errors are occluded by the markers, so i've zoomed in on the area with the worst discrepancy (highest variance) in Figure 2. Even there, the agreement is quite good! Finally, I compared the sample variance of the 1000 samples to the analytic sample variance for truncated ℓ given in Equation (171), truncating ℓ at $\ell_{\max}$ in the sums. This is shown (without standard error) in Figure 3. **Note: I haven't yet implemented the "add back" of the droop in the sample mean. This should be pretty quick to do though.** These test computations were done with the `Test.py` script, which provides a
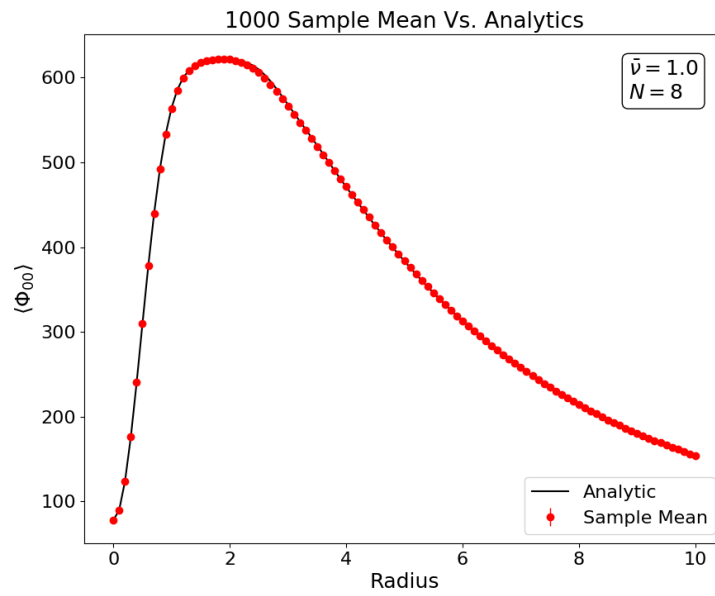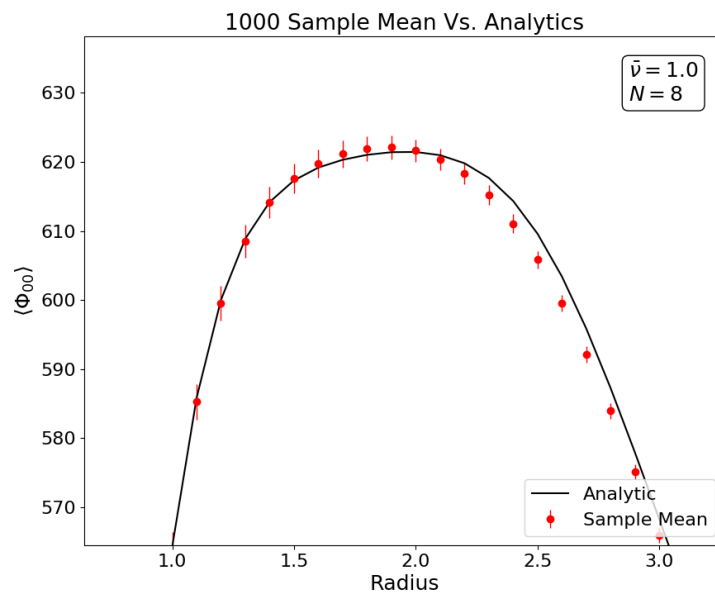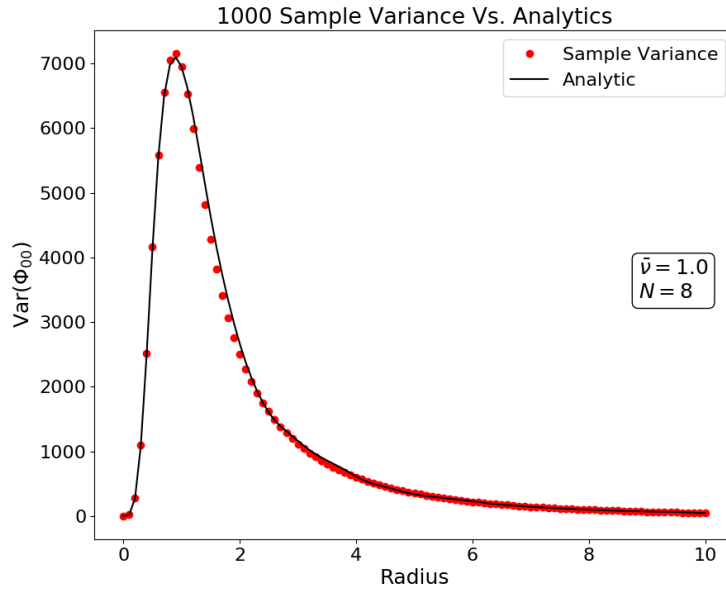
Figure 1:

Figure 2:

Figure 3:

simple implementation of the classes described here, and `Analysis.py` which is essentially just doing the plotting.