Ariel Barnea
June 2023
Argonne National Laboratory
Final Bispectrum Report

Table of Contents

1 Introduction

1.1 What is the Bispectrum

The Power Spectrum has long served as a reliable tool for analyzing the statistical properties of cosmic structures. In a Gaussian universe, it summarizes all of the essential information necessary to characterize the distribution of power across different angular scales. However, in our real universe, additional complexities arise, driven by gravitationally-induced non-Gaussianity. To extract a more comprehensive understanding of the underlying physics encoded in cosmological maps, we must turn to the Bispectrum.

The Bispectrum is a higher-order statistical measure that, by examining the correlations among three different angular modes, allows us to unravel non-Gaussian signatures that the Power Spectrum cannot. When seeking to understand the primordial universe and the formation and evolution of cosmic structures, understanding the presence of non-Gaussianities is crucial. Furthermore, the Bispectrum's unique property of incorporating three factors, compared to the two factors found in the power spectrum, provides an opportunity to break degeneracies that might be present in the latter. By scrutinizing the Bispectrum's additional degree of freedom, we can disentangle these degeneracies and gain a deeper understanding of the underlying physical phenomena.

As such, by exploring the Bispectrum, and specifically doing so in harmonic space, we can seek to understand a small subsegment of the the complexities of our universe, and hopefully learn a bit on how cosmic evolution operated.

1.2 The Purpose of this Report

I am writing this report to serve as a guide for any future researchers interested in the Bispectrum and non-Gaussianities. It is intended to explain all of the code I have written and the plots I have created during my time working on the Bispectrum project at Argonne National Laboratory. I also seek to explain the methodology and process behind the work of this project, so that any future researchers won't fall into some of the issues that we ran into and had to troubleshoot and fix. This guide also seeks to compile much of the theory, equations, and papers that underly the Power Spectrum and gravitationally-induced weak-lensing Bispectrum into one big guide that is readily accessible. Hopefully, this will also allow for a more bite-sized approach to understanding the maths behind the Bispectrum, as my goal is to compile this information in a more readable way. Finally, this guide also serves as a place to discuss some of the computational methods that I have learned in my time at Argonne. I seek to do so in a way that will help future researchers not have to go searching through the depths of Python package documentations and YouTube tutorials like I had to, and instead be a personal guide to understanding the underlying infrastructure and applications of these computational methods. I conclude this report with a brief discussion on the possible directions for future research with this project and Bispectrum code, along with some thoughts on what I learned from this project and my time working for the Computational Physics and Advanced Computing Group at Argonne National Laboratory, and a final acknowledgements (i.e. thank you's) to people that helped me along the way.

2 Methodology and Process

In this section, I discuss the theory behind the two main methods that we utilized when creating the Bispectrum calculator. This is in Section 2.1. Then, in Sections 2.3 and 2.2, I go into some of the challenges that we ran into and the lessons we learned from trying to speed up the code from its prototype version.

2.1 Theory Calculations and Equations

This subsection is intended to give an overview / compile all of the different theory calculations, equations, and information that others might find useful to have all in one place, if you are trying to delve into the world of Bispectrum calculations.

2.1.1 Filter Average Method

The first method I will go into is the filter method, pulled from [1], which seeks to estimate the value of the bispectrum (and later, binned bispectrum). This method is primarily focused on calculating the individual $B_{\ell_1\ell_2\ell_3}$ values of the bispectrum and averaging them to find the overall averaged bispectrum. The equation for doing this is:

$$B_{\ell_1\ell_2\ell_3} = \sqrt{N_\Delta^{\ell_1\ell_2\ell_3}} \sum_{m_1,m_2,m_3} \begin{pmatrix} \ell_1 & \ell_2 & \ell_3 \\ m_1 & m_2 & m_3 \end{pmatrix} \langle a_{\ell_1 m_1} a_{\ell_2 m_2} a_{\ell_3 m_3} \rangle \tag{1}$$

where the normalization factor is given by:

$$N_\Delta^{\ell_1\ell_2\ell_3} = \frac{(2\ell_1+1)(2\ell_2+1)(2\ell_3+1)}{4\pi} \begin{pmatrix} \ell_1 & \ell_2 & \ell_3 \\ 0 & 0 & 0 \end{pmatrix}^2 \tag{2}$$

Here, the quantity $\langle a_{\ell_1 m_1} a_{\ell_2 m_2} a_{\ell_3 m_3} \rangle$ is the CMB angular bispectrum and $N_\Delta^{\ell_1\ell_2\ell_3}$ can be thought of as the number of possible (ℓ_1, ℓ_2, ℓ_3) triangles on the celestial sphere. Note the use of the Wigner-3j values (see Section 2.3) here. Also note that we excluded the polarization factors because we're dealing with scalar temperature fields. Further important information to note about these equations include that:

1. The bispectrum is symmetric under the simultaneous exchange of its three multiple numbers ℓ_1, ℓ_2, ℓ_3. Basically, we can just restrict our computation of $B_{\ell_1\ell_2\ell_3}$ to ℓ values s.t. $\ell_1 \leq \ell_2 \leq \ell_3$.
2. The triangular condition must be satisfied, i.e. $\ell_1 + \ell_2 + \ell_3 =$ constant.
3. The parity condition must be satisfied, i.e. $\ell_1 + \ell_2 + \ell_3 =$ even.
4. The triangle inequality is satisfied, i.e. $|\ell_1 - \ell_2| \leq \ell_3 \leq \ell_1 + \ell_2$.

Conditions 3 and 4 come from the fact that the m values in the normalization factor are all equal to 0. These conditions are very useful, and I would recommend becoming familiar with them as they speed up maths and computations.

We also went into the theoretical variance for this calculation, which is given by:

$$\text{Var}\left(B_{\ell_1\ell_2\ell_3}\right) = g_{\ell_1\ell_2\ell_3} N_\Delta^{\ell_1\ell_2\ell_3} C_{\ell_1} C_{\ell_2} C_{\ell_3} \equiv V_{\ell_1\ell_2\ell_3} \tag{3}$$

such that $N_\Delta^{\ell_1\ell_2\ell_3}$ is as before, namely:

$$N_\Delta^{\ell_1\ell_2\ell_3} = \frac{(2\ell_1+1)(2\ell_2+1)(2\ell_3+1)}{4\pi}\begin{pmatrix}\ell_1 & \ell_2 & \ell_3\\ 0 & 0 & 0\end{pmatrix}^2 \tag{4}$$

and $g_{\ell_1\ell_2\ell_3}$ is equal to either 6, 2, or 1 depending on whether 3, 2, or no ℓ's are equal respectively. This implies that, for the equilateral case we're dealing with above, $g_{\ell_1\ell_2\ell_3} = 6$, giving us a variance:

$$V_{\ell_1\ell_2\ell_3} = 6 \cdot N_\Delta^{\ell_1\ell_2\ell_3} \cdot C_{\ell_1}C_{\ell_2}C_{\ell_3} \tag{5}$$

It should be noted that, for the binned bispectrum:

$$B_{i_1i_2i_3} = (\Xi_{i_1i_2i_3})^{-1}\sum_{\ell_1\in\Delta_1}\sum_{\ell_2\in\Delta_2}\sum_{\ell_3\in\Delta_3} B_{\ell_1\ell_2\ell_3} \tag{6}$$

where $\Xi_{i_1i_2i_3}$ is the number of ℓ triplets within the (i_1, i_2, i_3) bin triplet satisfying the triangle inequality and parity condition selection rule, the variance $\mathrm{Var}(B_{i_1i_2i_3})$ is:

$$\mathrm{Var}(B_{i_1i_2i_3}) = \frac{g_{i_1i_2i_3}}{(\Xi_{i_1i_2i_3})^2}\sum_{\ell_1\in\Delta_1}\sum_{\ell_2\in\Delta_2}\sum_{\ell_3\in\Delta_3} N_\Delta^{\ell_1\ell_2\ell_3}C_{\ell_1}C_{\ell_2}C_{\ell_3} \equiv V_{i_1i_2i_3} \tag{7}$$

2.1.2 Map Method

The second method I will go into is the map method, also pulled from [1]. This also seeks to estimate the value of the bispectrum, but utilizes a map-based approach instead of a computation, filter averaged method.

This method stems from the straightforward computation of evaluating the integral over the sky:

$$B_{\ell_1\ell_2\ell_3}^{p_1p_2p_3,\mathrm{obs}} = \int \mathrm{d}\hat{\Omega}\, M_{\ell_1}^{p_1,\mathrm{obs}}(\hat{\Omega})M_{\ell_2}^{p_2,\mathrm{obs}}(\hat{\Omega})M_{\ell_3}^{p_3,\mathrm{obs}}(\hat{\Omega}) \tag{8}$$

where the ℓ triplets here also satisfy the selection rules from the Filter Method. This computation would take a lot of time if not for the fact that binning leads to a massive reduction in computational cost with minimal resolution loss. To accomplish this, we divide the ℓ-range $[\ell_{\min}, \ell_{\max}]$ into subintervals denoted by $\Delta_i = [\ell_i, \ell_{i+1} - 1]$ where $i = 0, \ldots, (N_{\mathrm{bins}} - 1)$ and $\ell_{N_{\mathrm{bins}}} = \ell_{\max} + 1$, so that the filtered maps are:

$$M_i^p(\Omega) = \sum_{\ell\in\Delta_i}\sum_{m=-\ell}^{+\ell} a_{\ell m}^p Y_{\ell m}(\hat{\Omega}) \tag{9}$$

and we use these instead of M_ℓ^p in our expression for $B_{\ell_1\ell_2\ell_3}^{p_1p_2p_3,\mathrm{obs}}$. The binned bispectrum is thus:

$$B_{i_1i_2i_3}^{p_1p_2p_3,\mathrm{obs}} = \frac{1}{\Xi_{i_1i_2i_3}}\int d\hat{\Omega} M_{i_1}^{p_1,\mathrm{obs}}(\hat{\Omega})M_{i_2}^{p_2,\mathrm{obs}}(\hat{\Omega})M_{i_3}^{p_3,\mathrm{obs}}(\hat{\Omega}) \tag{10}$$

where $\Xi_{i_1i_2i_3}$ is the number of ℓ triplets within the (i_1, i_2, i_3) bin triplet satisfying the triangle inequality and parity condition selection rule. The authors of this paper explain that $B_{i_1i_2i_3}^{p_1p_2p_3}$ can thus be thought of as an average over all valid $B_{\ell_1\ell_2\ell_3}^{p_1p_2p_3}$ in the bin triplet.

This method allows for a relatively straightforward and quick computation of $B_{i_1i_2i_3}$ from $a_{\ell m}$, as we will see in Section 3.

2.1.3 Convergence Bispectrum

Most recently, we worked on finding the theoretical convergence bispectrum found in [7]:

$$B_{\ell_1,\ell_2,\ell_3} = \int_0^{\chi_s} d\chi \left[\frac{3\Omega_M H_0^2}{2a(\chi)}\right]^3 \chi^2 W^3(\chi,\chi_s) B_\delta\left(\frac{\ell_1}{\chi},\frac{\ell_2}{\chi},\frac{\ell_3}{\chi},\chi\right) \tag{11}$$

such that:

$$k_i = \frac{\ell_i}{\chi} \tag{12}$$

where $B_\delta(\mathbf{k}_1,\mathbf{k}_2,\mathbf{k}_3;\chi)$ is given by:

$$B_\delta(\mathbf{k}_1,\mathbf{k}_2,\mathbf{k}_3;\chi) = 2F_2(\mathbf{k}_1,\mathbf{k}_2,z)P_\delta(\mathbf{k}_1,z)P_\delta(\mathbf{k_2},z) + \text{cyc. perm} \tag{13}$$

and

$$F_2(\mathbf{k}_1,\mathbf{k}_2,z) = \frac{5}{7}a(k_1,z)a(k_2,z) + \frac{1}{2}\frac{k_1^2+k_2^2}{k_1k_2}b(k_1,z)b(k_2,z)\cos\theta + \frac{2}{7}c(k_1,z)c(k_2,z) \tag{14}$$

where we're using $a = b = c = 1$, giving us:

$$F_2(\mathbf{k}_1,\mathbf{k}_2,z) = \frac{5}{7} + \frac{1}{2}\frac{k_1^2+k_2^2}{k_1k_2}\cos\theta + \frac{2}{7} \tag{15}$$

Which is needed for every cyclical permutation. We also require the lensing kernel $W(\chi,\chi_s)$, which is given by:

$$W(\chi,\chi_s) = \frac{\chi_s-\chi}{\chi\chi_s}\Theta(\chi_s-\chi) \tag{16}$$

where the step function $\Theta(\chi_s-\chi)$ is there to ensure that the lensing can't have a contribution from before the source existed. We can control this via integration, so we'll let $\Theta(\chi_s-\chi)=1$, leaving us with:

$$W(\chi,\chi_s) = \frac{\chi_s-\chi}{\chi\chi_s} \tag{17}$$

Note that, for the CMB, the redshift is just $z = 1089$. Implementing all of this together yields the convergence bispectrum for a set of ℓ_1, ℓ_2, ℓ_3 values, and can be done for ℓ-bins to find the binned convergence bispectrum for some map.

2.1.4 Bispectrum Templates

A brief tangent I went down whilst writing my initial bispectrum function is the different local vs. non-local bispectrum templates. More information about these Bispectrum templates and non-Gaussianities overall can be found in [1], which was instrumental in helping me understand these topics. For local f_{NL} potential, we have:

$$\phi_{\text{NL}} = \phi_{\text{Gauss}} + f_{\text{NL}}\left(\phi^2_{\text{Gauss}} - \langle\phi_{\text{Gauss}}\rangle^2\right) \tag{18}$$

For g_{NL}, we have:

$$\phi_{\text{NL}} = \phi_{\text{Gauss}} + g_{\text{NL}}\left(\phi^3_{\text{Gauss}}\right) \tag{19}$$

And for non-local f_{NL}, we have:

$$\phi_{\text{NL}} = \phi_{\text{Gauss}} + f_{\text{NL}}\left(\partial\phi^2_{\text{Gauss}} - \langle\partial\phi_{\text{Gauss}}\rangle^2\right) \tag{20}$$

2.1.5 Some Power Spectrum Notes

As a brief aside, I will discuss later some of the code I wrote for computing and plotting Power Spectra, so I will use this short section to detail the finite and infinite spherical shell cases for computing the Power Spectrum. More information about these cases can be found in [9].

The equation for finding C_ℓ from $P(k)$ assuming we're *not* dealing with a finite spherical thin shell is given by:

$$C_\ell = \frac{2}{\pi}\int_0^\infty dk \cdot k^2 \cdot P(k) \cdot \left|\frac{\Theta_\ell(k)}{\delta(k)}\right|^2 \tag{21}$$

However, noting that the plots we usually work with are already overdensity plots, implying that $\Theta_\ell(k) = \delta(k)$, we find:

$$\left|\frac{\Theta_\ell(k)}{\delta(k)}\right|^2 = \left|\frac{\delta(k)}{\delta(k)}\right|^2 = 1 \tag{22}$$

Therefore, for the ideal case, we find:

$$C_\ell = \frac{2}{\pi}\int_0^\infty dk \cdot k^2 \cdot P(k) \tag{23}$$

Now, when we're dealing with a finite spherical thin shell, we have to amend our previous equation with a `sinc` correction factor. This, we find a C_ℓ expression for the finite case of:

$$C_\ell^{\text{shell}} = \frac{1}{r^2}\int \frac{dk_\parallel}{2\pi} \cdot P_\delta(k) \cdot \text{sinc}\left(\frac{\mathrm{k}_\parallel \Delta \mathrm{r}}{2}\right) \tag{24}$$

such that $k = (k_\parallel^2 + |\mathbf{k}_\perp|^2)^{1/2}$, where $\mathbf{k}_\perp = \ell/r$. However, in the case of the infinite shell, we can just take the limit as $k_\parallel \Delta r \to \infty$ of the thin spherical shell, which would give us:

$$\lim_{k_\parallel \delta r \to \infty} C_\ell^{\text{shell}} = \frac{1}{r^2 \Delta r} P_\delta\left(k = \frac{\ell}{r}\right) \tag{25}$$

2.2 Numba

Numba is a Just-In-Time (JIT) compiler for Python that specializes in optimizing and accelerating numerical computations. If you are a Python-only coder like I was when I first started using Numba and don't understand what this means, don't worry, I will explain it here! I will also explain how to use Numba, and why you should use Numba for computationally expensive and *repetitive* tasks that don't involve sharing memory allocation.

First, a brief explanation of why we use Numba. Python is a very convenient, simple, and easy-to-use language with a lot of capabilities. You don't have to declare types, worry about pointers, allocate and free memory, etc. However, to allow the users (us) to get away with all of this, Python has to do all of this stuff internally. For example, declaring $i = 0$, you don't have to state that i is an integer or unsigned integer, as Python will do this for you by creating an integer object with a value of 0 and a "reference" i. But to access i, you have to access an object, not just straight bytes (i.e. a bunch of 0 and 1 bits). Creating and appending to lists means that Python has to allocate a bunch of space for the list, way more than you will probably need, and then tie the values in your list together every time you append. So basically, this makes things very convenient for you

as the user, but not very computationally efficient, as it has to go through the Python interpreter and deal with all of this convenience.[1]

Numba bridges this gap with Python by compiling your Python code straight into highly-efficient machine code using just-in-time compilation techniques. Using the `@jit` function decorator (I will explain this shortly) on a function allows Numba to analyze the function code and apply various optimizations. It inspects the types of the input arguments and generates optimized machine code specific to those types. This process is known as type specialization. `@njit` is another Numba function decorator which stands for `No Python jit`, which is basically just "jitting" a function but without going through the Python interpreter. It is very important to note that jitting a function means you lose a lot of Python's capabilities (i.e. calling complex functions that aren't also jit-compatible, using Python data structures like lists and dictionaries, etc.). That's why jitting is so useful for repetitive computations like loops, where C shines, but not used for things like plotting, where Python shines. Since computing the Bispectrum, specifically the filter average method from Section 2.1.1, requires a lot of looping and summing and not many Pythonic data structures and functions, this makes our Bispectra calculations a prime candidate for Numba. Hence why this section exists. However, before I get into how I used Numba, I want to talk briefly about parallelization.

Another very useful element of Numba is that you can parallelize your code across multiple threads. I primarily used the parallel loops functionality of Numba, as most of my Bispectrum calculations involved two or three nested for loops at a time (for example, when summing over all of the m values in an ℓ_1 and ℓ_2 range, or summing over all of the ℓ-values in an i-bin). Basically what this means is that I split up my computation of $B_{\ell_1\ell_2\ell_3}$ across multiple *threads*. Threads are basically just independently run portions of a program that can be spread across multiple cores of your computer and run simultaneously. When you launch n threads, you can think of your computer distributing the function into n different parts and running them all concurrently. For example, if you want to thread a function that sums all the values from 1 to 1600 across 16 threads, you can distribute the summing of the values 1 to 100 to thread 1, values 101 to 200 to thread 2, 201 to 300 to thread 3, etc.[2] This is a very useful concept in Computer Science, but not one I am going to flesh out today. Instead, I am going to talk about Numba's `parallel=True` capabilities when in a jitted function. Activating this flag will basically let Numba know that you want to distribute the function's job across multiple threads, and it will do it automatically across the number of threads you provide (normally, you have to manually do this). This automatic threading only works for *loops* (since it's simple enough without having to do it manually), but conveniently, that's exactly what we need it for. So, when summing up Bispectrum values[3], you can parallelize the for loops across n threads to speed up your computation by *about* n times.[4]

Okay, now I will very briefly explain how to use Numbas capabilities. Basically, whenever you want to jit a function, first make sure it can be jitted (otherwise Numba will give you an error), which you can do by checking if you have any non-jitted function calls or are using a bunch Pythonic data

[1] This is why C is extremely speedy, because it compiles straight down into assembly and then machine code, and doesn't have to go through all of these hoops to operate — but it requires a lot more strict control, typesetting, and memory management.

[2] This is not a good way to distribute this computation by the way, I'm just using this as an example.

[3] Notice that this does not involve storing any values anywhere, which is why this works.

[4] Overhead costs of initializaing your threads and passing in the necessary parameters can slow this down.

types in your function like dictionaries, lists, and/or objects. Then, you just add the jit function decorator `@njit` to the line before the function declaration, like:

```
@njit
def foo(x: int) -> int:
    return x
```

And that's it, your function is now jitted in no Python mode. To parallelize a loop specifically, you can just add the flag `parallel=True` to your function decorator, and then use a function like `prange` on a loop instead of `range`. For example:

```
@njit(parallel=True)
def bar(n: int) -> int:
    sum = 0
    for i in prange(n):
        sum += i
    return sum
```

`prange` basically just distributes the job of looping across multiple threads. If you want to set the number of threads, you can just include the function `set_num_threads(n_threads)`, where `n_threads` is some integer number of threads, somewhere in the function, and that's it! For example:

```
@njit(parallel=True)
def bar(n: int, num_threads: int) -> int:
    sum = 0
    set_num_threads(n_threads)
    for i in prange(n):
        sum += i
    return sum
```

And don't forget to import all of these functions, by the way.

```
from numba import jit, njit, prange, set_num_threads
```

Congratulations, that's basically everything you need to know about Numba for the purposes of Bispectrum computations!

An Aside: Numba Typing

If you want to used lists and dictionaries with jitted functions, look into Numba's Typed libraries to declare these Numba-specific types. Basically, it just declares a specific type for the entire data structure so that Numba can parse it. You can find more information in Numba's documentation.

2.3 Wigner-3j

The Wigner-3j coefficients play a big role in many Bispectrum computations, so I am using this section to briefly discuss how we used them, the (many) issues we encountered with them, and how you can hopefully not have to deal with these because of this guide!

There were two main packages we used to compute the Wigner-3j coefficients, each with their benefits and drawbacks. I will summarize them both and explain which one we ended up using.

2.3.1 Spherical Wigner-3j Calculator

The first Python package I used to compute the Wigner-3j coefficients was `spherical`.[5] Spherical is specifically designed for evaluating and transforming Wigner's $\mathcal{D}$ matrices, Wigner's 3-j symbols, and spin-weighted (and scalar) spherical harmonics. I believe it relies on a recursive equation set to calculate the w3j values, and it does so very quickly. It also works in combination with numba's just-in-time environment, which is extremely useful for the filter average method of computing the Bispectrum, as explained in Section 2.1.1.

This was my package of choice at the start. However, this package has two glaring issues that proved fatal.

1. **It can't compute Wigner-3j values for high ℓ, which I found to be somewhere around $\ell \approx 1000$.** I don't know why it can't, even after searching through the code to figure it out. It doesn't seem like the authors are aware of this issue, likely because they didn't use it to compute ℓ values on the order of $\ell \sim 10^3$. If you try to compute ℓ values greater than this range, the computation simply returns `nan`. Obviously, this is a very big issue for Bispectrum calculations, which can have high angular sky resolutions. As I mentiond, combing through the `spherical` package did not prove useful in figuring out what was wrong, and I couldn't find any documentation online either of this issue.

2. **It occasionally has sign issues in computation which can catastrophically skew overall Bispectra computations.** This was really bizarre. Sometimes, just the signs would be off, usually for lower ℓ values, and sometimes, the values would be orders of magnitude off, usually for high-ish ℓ values. I only realized this *after* we switched to a different Wigner-3j package (see next subsubsection) and noticed a different in w3j values for the same ℓ and m values. To determine correctness, I compared `spherical` to `pywigxjpf`, and used `SymPy` (these packages are the subject of the next two subsubsections) as an intermediary for confirmation; that's how I discovered this weird sign issue. We believed that the sign issues for lower ℓ computations caused the large skewedness for higher ℓ computations.

These two issues, and the inability to figure out what was wrong with `spherical`, led us to a point where we were ready to recode the Wigner-3j methods of `spherical` ourselves. However, luckily, Salman knew of a different, albeit slower, package to compute Wigner-3j coefficients, which is what the next subsubsection will discuss: `pywigxjpf`.

2.3.2 pywigxjpf Wigner-3J Calculator

The second, and current, Python package I used in my Bispectrum computations is `pywigxjpf`. This is a Python wrapper for the `wigxjpf` package, which I believe is coded in C. This package is specifically designed to evaluate Wigner 3j, 6j and 9j symbols using prime factorization and multi-word integer arithmetic. More information can be found in [4]. This package has a higher barrier to entry to use than `spherical` and is also a lot slower than `spherical`, but has the very big benefit that it computes the Wigner-3j coefficients *accurately* and for *all* ℓ *values*, including $\ell \gtrsim 1000$, as confirmed by `SymPy` and `WolframAlpha`. I will explain some of these notes now.

The first big note I will mention about `pywigxjpf` is that **all ℓ and m arguments must be**

[5] https://pypi.org/project/spherical/.

doubled when inputting them into the `wig` function. I'm sure there is an explanation of why this is somewhere in the documentation, but I didn't really bother looking.

The second, more important note is that this package is *a lot* slower than `spherical` and does not work with Numba by default. Testing `spherical` against `pywigxjpf` showed a 500x speed up for the former over the latter. The big fatal drawback of course being that `spherical` doesn't compute the correct values. As such, even though it's slower, `pywigxjpf` was our package of choice, and what we use now. However, there is a way to get it working with numba, which is what I am to briefly explain now.

To get `pywigxjpf` working with numba, you basically have to use this weird wrapper function that I don't fully understand. It is called `nb_wig3jj` and you have to import it using `cffi`. I've put the code here:

```
from numba.core.typing import cffi_utils as cffi_support

try:
    from pywigxjpf_ffi import ffi, lib
except ImportError:
    from pywigxjpf.pywigxjpf_ffi import ffi, lib

cffi_support.register_module(pywigxjpf_ffi)

nb_wig3jj = pywigxjpf_ffi.lib.wig3jj
```

Now, you can call `nb_wig3jj` as normal (don't forget to double all inputs!). However, you will also need to allocate and free table and temp memory. Table memory can be allocated as follows:

```
lib.wig_table_init(val_init, 3)
```

where `val_init` is just the largest $2\ell + 1$ value you are going to use, and the 3 indicates that we're computing Wigner-3j values (use 6 or 9 for Wigner-6j and -9j respectively). You then also have to allocate temp memory:

```
lib.wig_temp_init(val_init)
```

where `val_init` is the same as before. You have to allocate the table before a set of computations, and temp for *each* computation. Once you are done with each computation, make sure you free the temp memory allocation using:

```
lib.wig_temp_free()
```

And once you are done with the full set of Wigner-3j computations, you can free the table memory using:

```
lib.wig_table_free()
```

I believe this freeing comes from the underlying C architecture. And that's how you use `pywigxjpf`! You can find a full breakdown of all of this, as well as examples, in the `w3j_threading_test.ipynb` notebook in my Bispectrum repository on Github (see Section 3). I included the explanation of the Wigner-3j code here because I believe this is a required methodology step to get the code working, instead of new code I wrote.

2.3.3 An Aside: SymPy

SymPy is a Python package for symbolic mathematics and has a Wigner-3j calculator built in as well. It does most of its computation in underlying algebra, and isn't good for quick computations. However, I used it occasionally to confirm Wigner-3j calculations in Spherical and pywigxjpf, as I mentioned earlier. More information about SymPy can be found in [6].

3 Codes

The purpose of this section is to go over some of the finalized Power Spectrum and Bispectrum code I've been working whilst at Argonne. I am framing this section more in the sense of showing the topics of the work, and what would be most useful for someone in the future who is reading this to know about the code. All of my Bispectrum code can be found in the following Github repository:

Github: https://github.com/abarnea/Bispectrum

Please note that nearly all of the final-product functions, notebooks, and Python files in this repository are documented with docstrings and typesetting, so you can navigate and explore these functions yourself in the Git repository.

3.1 Power Spectrum

The theory behind the power spectrum code can be found in Section 2.1.5, and the first real set of computations and graphs can be found in the `Power_Spectra_LJ.ipynb` Jupyter notebook, which was tested on a Last Journey (LJ) density map (see [3] for more information about LJ). The other two Jupyter notebooks in the `power_spectra` folder are initial drafts of HEALPix Power Spectra use cases using the `anafast` method. These could be useful for some initial understanding of how the Power Spectrum, HEALPix, and Anafast work — I would specifically look at the `Anafast_Power_Spectrum.ipynb` notebook. Also, it is important to note that many of the functions worked on and used in `Power_Spectra_LJ.ipynb` are located in the `helper_funcs.py` Python file.

One of the biggest takeaways from `Power_Spectra_LJ.ipynb` and `helper_funcs.py` are the comparison of HEALPix's `anafast` method to compute the Power Spectrum to the analytical computation using the $a_{\ell m}$'s. Reading through this notebook should hopefully give you a better understanding of how the C_ℓ's work as the square magnitude of the $a_{\ell m}$'s, and how the power spectrum as a distribution of power across a frequency range is computed in harmonic space using ℓ's. Note some key functions in the `helper_funcs.py` file, such as `compute_cls` and `sort_alms`, which are used in this notebook. Some of the data is pickled for faster loading, but the actual code used to get that pickled data is just commented out, so you can read it that way. Also note the use of some of the Power Spectra plotting functions from `helper_funcs.py`, specifically `plot_cl`. This proved useful for visualizing the Power Spectra, of course, but also just writing these functions and understanding how they plot the Power Spectrum in ℓ-space is helpful for understanding.

Another of the biggest takeaways from `Power_Spectra_LJ.ipynb` and `helper_funcs.py` are the computation of C_ℓ for the power spectrum $P(k)$ for both a thin finite spherical shell and infinite

shell. Again, the theory for this can be found in Section 2.1.5. Reading through this computation, along with the Core Cosmology Library (CCL) power spectrum utilization and computation, will provide better insight into how the power spectrum interacts with the $a_{\ell m}$'s in harmonic space. See [2] for more information about CCL. This proved very helpful for my understanding prior to delving into the more complicated world of the Bispectrum. The key function that does this is `pk2cls`, which relies on `_cl_shell`. Again, reading through this is a fairly straightforward implementation of the theory for the C_ℓ computation in both the finite and infinite cases. The key result, i.e. plot, comes from `LJ Power Spectra` plot, which compares the Anafast C_ℓ's, Computed C_ℓ's, C_ℓ's from $P(k)$ for the finite thin spherical shell, and the $C_\ell's$ from $P(k)$ for the infinite shell. This plot will be discussed further in Section 4.1.

Finally, one quick aside I wanted to mention is the `sort_alms` function, which can be found in `helper_funcs.py`. I mention this because the HEALPix output for the $a_{\ell m}$'s from a map return a one-dimensional numpy array, where the $a_{\ell m}$'s are ordered by m value, *not* ℓ value. For example, all the $m = 0$ $a_{\ell m}$ values are first, then all the $m = 1$ $a_{\ell m}$ values, etc. There are no negative m values because HEALPix automatically adds them in when using HEALPix functions — but this isn't useful when you want to work with the $a_{\ell m}$'s of a map alone. Therefore, I wrote up this function, `sort_alms`, which parses through this HEALPix numpy array of $a_{\ell m}$'s and puts them in a sorted dictionary keyed by ℓ values, where each element is a staggered numpy array of the m values. This proved very helpful for my initial Power Spectrum and Bispectrum computations, and I would argue that it is important to understand if you want a more well-fleshed out understanding of HEALPix and the $a_{\ell m}$'s. The two drawbacks of this dictionary-sorted $a_{\ell m}$ function and data structure are:

1. It's difficult to access both individual $a_{\ell m}$ values and groups of $a_{\ell m}$ values with different ℓ values. This means a lot of iteration through the dictionary, which is computationally expensive and inefficient compared to just storing everything in a one-dimensional numpy array like HEALPix does.

2. You can't use Numba on a dictionary, particularly one with different-sized Numpy arrays as the elements. This means that, in order to compute Bispectrum values using Numba, you have to loop through all the ℓ values the sorted $a_{\ell m}$ dictionaries and pull out each respective set of m's. This obviously does not allow us to use Numba to its fullest capabilities.

However, I would still argue that it was useful for me to understand, code up, and use this sorted $a_{\ell m}$'s dictionary at the beginning, as it gave me a better understanding of how the ℓ, m, $a_{\ell m}$, and C_ℓ values all worked in relation to each other, the Power Spectrum, and the Bispectrum. When we get to the next section, Section 3.2, I will discuss how I wrote a new function that directly filters the $a_{\ell m}$'s for the Map method of computing the Bispectrum, without using this `sort_alms` dictionary intermediary. You will also see how I got around the dictionary data structure for the Filter Average method of computing the Bispectrum.

3.2 Bispectrum Calculator

As explained in Section 2.1, there were two main methods we utilized to compute the Bispectrum — the filter average method and the map method. This section goes into the code for each. The final, compiled code for both methods can be found in `bispec_calculator.py`, along with the proper documentation via docstrings and typesetting.

3.2.1 Filter Average Method

We start with the Filter Average method for computing the gravitationally-induced weak-lensing Bispectrum, the theory of which can be found in Section 2.1.1, since this was the one I worked on first and for longer. This method very directly follows the theory, and mostly involves a lot of $a_{\ell m}$ configuring, Wigner-3j calcutions, and looping to manually compute the $B_{\ell_1\ell_2\ell_3}$. I would argue that this method is also more beneficial for educationally understanding what's going on with the Bispectrum.

The initial code for this method can be found in the `initial_bispectra_project` folder. The file `Bispectra_LJ.ipynb` contains the first draft of this code as it applies to the Last Journey density map, and the `Bispectra_Theory_Comparison.ipynb` tracks the development of this method's calculator. Some of the first few implementations, helper functions, and Numba-tests with this method can be found in the second half of the `helper_funcs.py` file as well. The final draft implementation of the Filter Average method can be found in the `bispec_alt_methods.ipynb` Jupyter notebook, with the final binned version being located in the `bispec_calculator.py` file.

I will briefly go over the process of implementing this method, and then go over some of the key points about the method. First, regarding the implementation, we ran into a bunch of issues that I will outline here, along with our solutions:

1. The first implementation worked, but was extremely slow, taking over 2 hours to do a basic computation up to $\ell \sim 50$. From there, we implemented Numba's `njit` capabilities into the function and added threading, as explained in Section 2.2, which substantially boosted performance. The process of dealing with this can be found in `Bispectra_LJ.ipynb` and `helper_funcs.py`.

2. We did a bunch of method testing with the Gaussian Bispectra of a theoretical density field, $C_\ell \sim \ell^{-3}$, plotted the Bispectra for many different maps, and analyzed the error spread and variances for this method on this field. The theory behind this can be found in Section 2.1.1, and the figures can be found in Section 4.2.2. This was important for us to test the Bispectrum calculator on a map whose results we already knew. This led us to use this field to pinpoint any issues with the overall behavior and functionality of the Bispectrum calculator. This entire process can be seen in the `Bispec_Theory_Comparison.ipynb` Jupyter notebook. From this testing, we recognized a number of issues — many of these were smaller coding-related issues, but a big one was in the functionality of the Wigner-3j calculations. The issues with this can be found in the enumerated item below.

3. Resulting from the testing in the last enumerated item, we recognized that the Wigner-3j calculator from the `spherical` didn't work for $\ell \gtrsim 1000$. We also later found out that the values of the calculator were occasionally incorrectly signed, causing major differences in the expected result. We thus switched to `pywigxjpf`, which we had to spend a while making Numba-compatible and dealing with the memory allocation issues that came with that. This process can be found in Section 2.3.

4. After completing the Map method Bispectrum calculator, the theory and code of which can be found in Sections 2.1.2 and 3.2.2 respectively, we found that the Bispectrum values for the same ℓ values and $a_{\ell m}$ values did not agree between the two methods. Comparing to WolframAlpha led us to discovere inaccuracies in the filter average method. The

`bispec_alt_methods.ipynb` notebook outlines the fixes to these inaccuracies, which came down to both convention differences between the papers we used to compile the this method and the computation methods. Those two changes can be found in `compute_bispec` function of this notebook and primarily were:

(a) Adding the `conv` factor into the expectation value of the $a_{\ell m}$'s computation, which is needed due to both a convention difference and a typing issue where the values weren't computed properly with numpy's complex float data type.

(b) Adding a permutation weighting to the final Bispectrum sum in addition to the normalization factor, which scaled the the result down to account for the even parity conditions that allowed us to ignore a bunch of the ℓ and m value loops. This permutation weighting can be found in the function `get_perm_weighting`, also in this notebook.

5. We didn't have any binning in the final Bispectrum function. The process of creating and selecting bins can be found in Section 3.2.2, but that is not the topic of this method. The final wrapping function, which created the filter average Bispectrum using the fixed `compute_bispec` function, is called `compute_average_bispec`, and can be found in the aforementioned Jupyter notebook as well.

Please note that there are a lot of different implementations of the filter average method for computing the gravitationally-induced weak-lensing Bispectrum scattered around the Github repository. The final ones to be concerned with are the implementation in ***`bispec_alt_methods.ipynb`*** *and the final version of the calculator in* ***`bispec_calculator.py`****.*

Now, I will take some time to walk through how the final version of the filter average method for computing the gravitationally-induced weak-lensing Bispectrum works, and how to use it.

1. The final implementation for this method can be found in the `bispec_calculator.py` file under the "## Filter Method" comment.

2. The bulk of the implementation for this method can be found in the `compute_bispec` function and the called helper functions like `compute_bispec_norm_factor`. It tracks pretty closely with the theory in Section 2.1.1, the Numba explanations in Section 2.2, and the Wigner-3j implementation in Section 2.3.2. A few key notes about this function:

   (a) This function and method overall uses the `sort_alms` dictionary and accesses $a_{\ell m}$ arrays via the ℓ key. This makes things more computationally expensive, as you have to first sort any $a_{\ell m}$'s, and then loop over all of the indices to access the numpy arrays of $a_{\ell m}$ values. It might be useful to explore if there is a way to do this without using the sorted $a_{\ell m}$'s dictionary, like what is possible for the Map method in Section 3.2.2.

   (b) Pay attention to the Wigner-3j function memory management allocation and frees when finding the Bispectrum Normalization Factor, which uses `nb_wig3jj`, and the Wigner-3j computation inside the nested loop. More information about how this works can be found in Section 2.3.2.

   (c) There are only two for loops looping over ℓ values, as having two m values automatically determines the third using the rule that $m_1 + m_2 + m_3 = 0$.

   (d) See enumerated item 4 from the outline of issues earlier in this section for further information about `conv` and `get_perm_weighting`.

This function computes the Bispectrum for a single set of ℓ-triplets, so it more serves as a helper function to compute the Bispectrum for a range of ℓ-triplets, which is described in the next enumerated item.

3. To find the averaged Bispectrum value, which is what the Filter Average method is meant to do, you call `compute_averaged_bispec` using three bins i_1, i_2, and i_2, along with a dictionary of sorted $a_{\ell m}$'s from the `sort_alms` function. It utilizes the `find_valid_configs` function to determine the valid triangles to loop over, and then finds the averaged Bispectrum value by summing over all of the ℓ-triplet configurations and calculating the individual Bispectrum value for each.

4. The remainder of the code in the Filter Method section of `bispec_calculator.py` is devoted to the computation of the gravitationally-induced weak-lensing Bispectrum variance for the unbinned and binned cases. The theory behind this can be found in Section 2.1.1, and the resultant figures when applied to a theoretical field $C_\ell \sim \ell^{-3}$ can be found in Fig. 9.

Overall, the Filter Average method provided a very good foundation to work from in understanding how to compute the Bispectrum and the interplay between the ℓ multipole triplets, the $a_{\ell m}$'s, and the Wigner-3j coefficients. It provided a good opportunity to learn more about Numba and optimizing computational efficiency as well. There were a lot of issues with this method, but it gave a solid foundation and a lot of good helper functions and plotting functions to work from. However, as we will see in the next section, while accurate, the Map method does a much better job of computing the gravitationally-induced weak-lensing Bispectrum.

3.2.2 Map Method

The Map method for computing the gravitationally-induced weak-lensing Bispectrum, the theory of which can be found in Section 2.1.2, provides a quicker way of finding the Bispectrum from a certain density map by leveraging ℓ-bin filtering and HEALPix methods. The process of creating the code can be found in the `bispec_alt_methods.ipynb` Jupyter notebook under the aptly-named "Map Method" section. The element that makes the Map method work is that we have to create filtered maps for bins of ℓ-triplets, denoted by the letter i. Filtering and integrating these maps over the sky, and then dividing by the number of valid triangles, gives us the binned Bispectrum estimator using the Map method. The code for doing this is fairly straightforward, but I will go into a few key points about this method here.

1. The first portion of the map method is filtering the $a_{\ell m}$'s according to the ℓ's in a given i-bin.To do so, I originally used the `sort_alms` function to sort the $a_{\ell m}$'s, then used a `filter_map_binned` function to filter the sorted_alms dictionary according to the bins, and then finally another `unsort_alms` function to turn the filtered $a_{\ell m}$ dictionary back into a Numpy array that can be inputted into HEALPix to generate a map (using the `hp.alm2map` function). However, this proved to be incredibly inefficient, and the longest part of a Bispectrum computation for the Map method. Thus, in the `bispec_reindexing.ipynb` Jupyter notebook, I set out to directly filter the HEALPix-outputted numpy arrays of $a_{\ell m}$'s from a given map. I was able to do this by taking advantage of HEALPix's vector operations to find $a_{\ell m}$'s, ℓ-values, and m-values. *Note: The vectorized nature of these functions was not properly documented in the HEALPix documentation, so there is some testing in this notebook*

to determine exactly how it functioned. The final testing version of this function can be found in said Jupyter notebook and is called `filter_alms_no_dict`, with the finalized version in the `bispec_calculator.py` file under the function name `filter_alms`. This directly filters a numpy array of $a_{\ell m}$'s into a numpy array that can be inputted into HEALPix's `hp.alm2map` function, which is then used in the map method Bispectrum computation.

2. The next portion of the map method is the binning process. The function for creating and selecting the correct bins can be found in the `bispec_calculator.py` file and is named `create_and_select_bins`. It calls the `_create_bins` function to actually make the bins, and then selects the relevant three bins based on a user-inputted list of indices. These bins are the ones then used to create the maps from which the map method then filters and computes the Bispectrum with.

3. The next portion of the map method is creating the maps and getting the pixel area. The code for the former can be found in the function `create_bins_and_maps`, which combines the creation and selection of bins from the `create_and_select_bins` function with the `hp.alm2map` HEALPix function to create the maps. Getting the pixel area can be found in the `get_pixel_area` function, and just uses HEALPix map functionality.

4. The second to last portion of the map method is checking the validity of certain triangle configurations — this is done in the `check_valid_triangle` function. This function can then be used for two main, and important, purposes:

   (a) To count the number of valid triangles in the sky — the code of which is found in the `count_valid_configs` function.

   (b) To determine whether a Bispectrum computation should be made for a given ℓ-triplet.

   Importantly, both `check_valid_triangle` and `count_valid_triangle` can be run in a No Python jit environment through Numba, with the latter having the ability to be auto-parallelized with `Numba.prange`. This makes computation especially quick for the normalization factor in front of the binned Bispectrum estimator computation in the map method.

5. Finally, the function `compute_binned_bispec` combines everything everything according to the map method formula from Section 2.1.2 and computes the binned bispectrum.

Overall, the efficiency of the Map method far surpassed the efficiency of the filter average method due to its usage of vector operations in numpy and HEALPix, as opposed to the nested-looping that occurs in the fitler average method, even with Numba parallelization.

3.2.3 Bispectrum Calculator Accuracy Notes

Overall, exploring and utilizing the Map method for computing the gravitationally-induced weak-lensing Bispectrum proved extremely useful not just for its computational efficiency, but because it provided us with a secondary test to check the accuracy of the filter method's Bispectrum calculations — which we were using the majority of the time before when calculating, comparing, and plotting Bispectra. Upon completion of the Map method, it actually led us to discover the scaling issue that was mentioned in Section 3.2.1, as the values were mismatched, and WolframAlpha agreed with the Map method Bispectrum value for all sets of ℓ's and m's. We were then able to go

back and fix said scaling issue — a result of mismatched conventions between papers — and find agreeable results. Therefore, having these two different methods for computing the gravitationally-induced weak-lensing Bispectrum is extremely important, as it allows us to confirm and compare the accuracy of our calculations.

3.2.4 Final Bispectrum Calculator Thoughts

I would say that the next thing to focus on for these methods is just cleaning up their interface a bit in order to make them easier to work with and use, i.e. like having an overall function that can pick between the methods, and takes in a simple set of inputs and can find the Bispectrum from there. Currently, using these functions can be a bit of a mess — for which the goal of this section is to give a better understanding of how all of this code works.

3.3 Convergence Bispectrum

The theory behind the convergence bispectrum code can be found in Section 2.1.3, and the code can be found in the Jupyter notebook `convergence_bispec.ipynb`. I will take this subsection to go over the process and functions in this notebook for future use. *Note: This notebook is still being tested to confirm accuracy, and binning still needs to be implemented.*

1. The core of this code is in the `interp_chi_from_z` function, which takes in a redshift z and creates both a χ and z interpolator. It relies on Scipy's `interpolate` package.

2. The other important part of this code is the `create_pk` function, which takes in a `kmax` parameter and information about the z-range. It then creates a cosmological model using CAMB and the standard cosmological parameters from Planck18, and initializes the matter power. It then creates CAMB's matter power interpolator using all of these parameters, which is crucial for the rest of the convergence bispectrum computation. For more information about CAMB, you can look at [5].

The rest of this code is generally pretty readable. It directly follows the theory layed out in Section 2.1.3. There are a few things to note:

1. I utilized Scipy's `quad` integration function to perform the χ integral, and the integrand of the function can be found in the private and aptly-named `_convergence_integrand` function.

2. Note the factor of c^2 in units of km/s in the dominator of the coefficient in the function `compute_convergence_bispec`. The theory section of the convergence bispectrum *does not* mention this because, by Cosmology convention, $c = 1$, but we still need to account for the proper unit conversions.

3. The `cumulative_convergence_bispec` function is where the actual convergence bispectrum computation occurs. Masking the internal computations as much as possible is a possible next step of this function, as this is just the first draft. It utilizes two `bispec_calculator` functions, namely `compute_bispec_norm_factor` function and `check_valid_triangle` functions (from `bispec`), as described in Section 3.2.

    (a) Binning has not been implemented into the convergence bispectrum notebook yet. However, it should be relatively straightforward to do so by using the `select_bins` function

from the `bispec` package. All that needs to be done is adjust the inputs and the way that the loops are iterated over, and that should be it. This is specifically for equilateral binning.

(b) Non-equilateral bispectrum computation has also not yet been implemented into the convergence bispectrum computation while we complete testing for the initial equilateral version.

At the bottom of this notebook, you can find the confirmation that interpolated Power Spectrum $P(k)$ aligns with the Core Cosmology Library (CCL) Power Spectrum. This provided us with a useful way of debugging and pinpointing the various scaling issues we ran into with the convergence bispectrum computations. More information about CCL can be found in [2].

4 Relevant Figures

This section presents some of the key figures from the various stages of the Bispectrum project. Each figure is divided into the relevant subcategory below, and accompanied by a figure caption explaining its results and relevance. To understand how these figures were found and plotted, see Sections 2.1 and 3 respectively.

4.1 Power Spectrum Plots

The first phase of the Bispectrum project was building the underlying Power Spectrum understanding and infrastructure. These key figures are some of the initial ones constructed during computation and comparison of the Power Spectrum. For the theory behind some of these plots, see Section 2.1.5, and for the code, see Section 3.1 for the code. The first two plots are for the theory $C_\ell \sim \ell^{-3}$ field used to compute and compare the initial Power Spectrum and Bispectrum. The last plot is the refined Power Spectra for a Last Journey (LJ) density map.

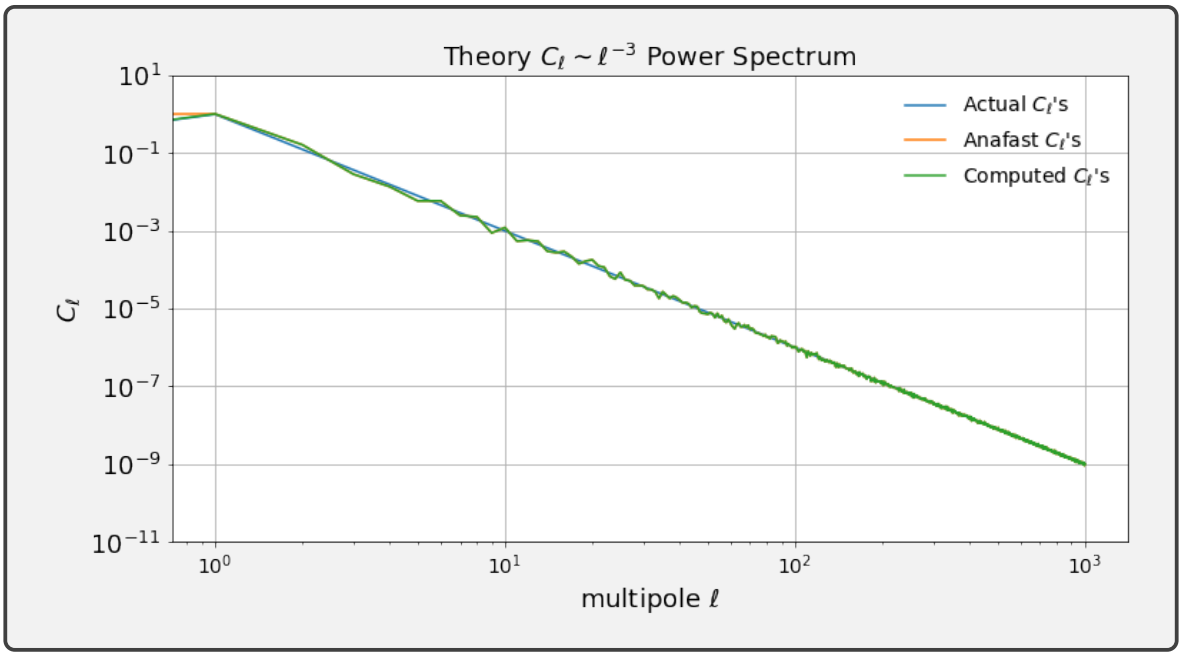

Figure 1: The figure above illustrates the theory $C_\ell \sim \ell^{-3}$ Power Spectrum using three different methods of computing the corresponding C_ℓ values. The actual C_ℓ's, constructed directly from the ℓ's are plotted in blue, HEALPix's `anafast`-constructed C_ℓ's are depicted in orange, and the manually-computed C_ℓ's from `compute_cls` can be found in green. From the plot, we can see that the actual and anafast C_ℓ's directly overlay each other, with the computed C_ℓ's providing a near direct overlay, with some background noise. This is particularly important since it demonstrates the ability to reconstruct the C_ℓ's manually, along with the accuracy of the helper functions to sort and compute $a_{\ell m}$'s and C_ℓ's. This provides useful for the resultant Bispectrum calculations later, as many of these underlying functions are used to compute the Bispectrum. Furthermore, some Bispectrum calculations, such as the Convergence Bispectrum, rely on accurate calculations of the Power Spectrum.

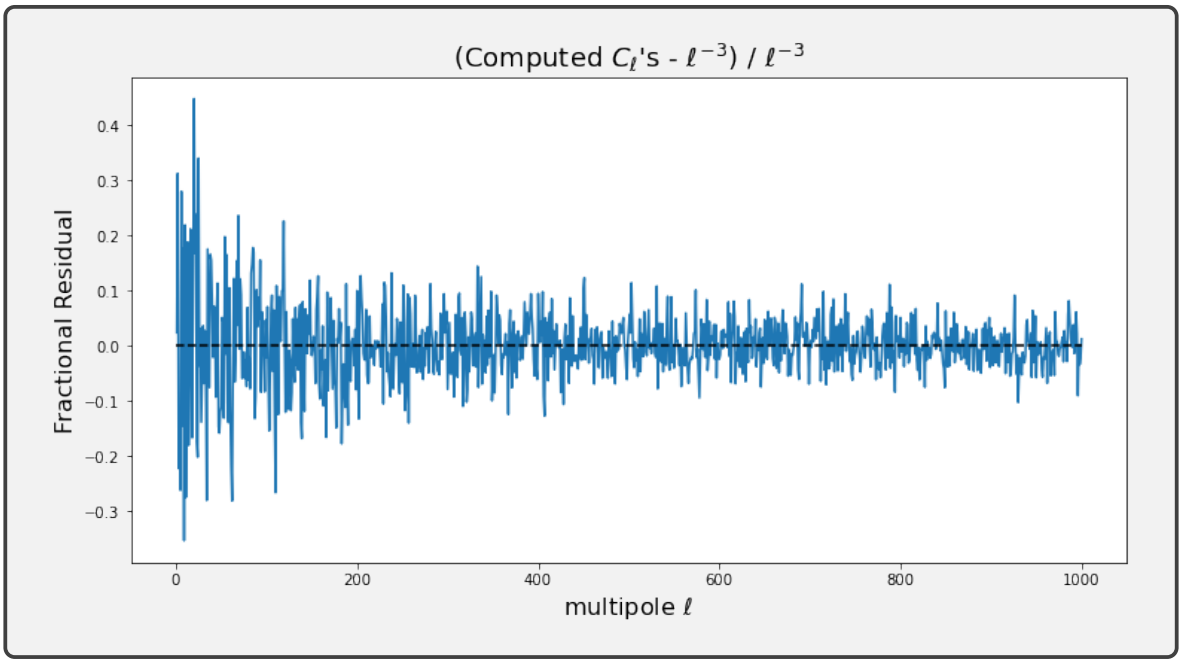

Figure 2: The figure above is a continuation of Fig. 1 and shows the fractional residual between the Computed C_ℓ's from the `compute_cls` function against the actual ℓ^{-3} field. That is, the noise that was discussed in that figure can be seen here. We can observe that the magnitude of the fractional residual is small, confirming our supposition early that the Computed C_ℓ's match closely with what they should be, and that the functions used to compute them are accurate for Bispectrum computations later.

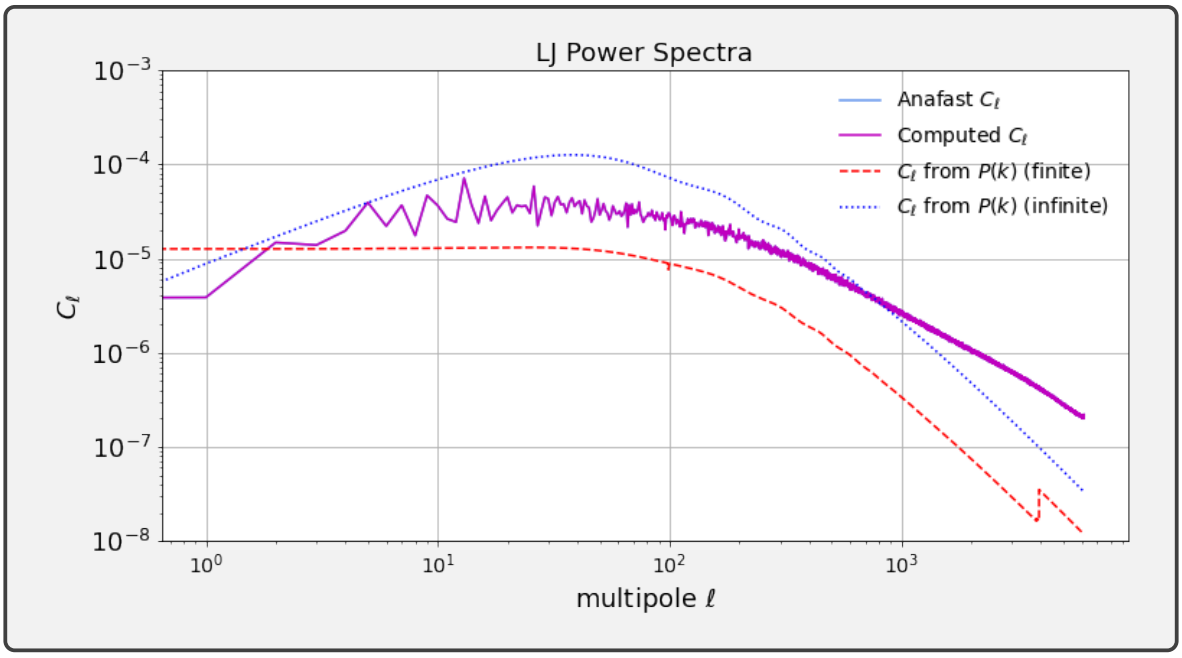

Figure 3: The figure above demonstrates the Last Journey (LJ) Power Spectra using two C_ℓ computation methods, `anafast` and manual-computation in solid blue and solid purple respectively, along with C_ℓ computation from $P(k)$ for the thin finite spherical shell and the infinite shell, in dashed red and dotted blue respectively. More information about Last Journey can be found in [3]. The first important resultant that we can see from this figure is that, on an actual density map, i.e. the one from Last Journey, the `anafast` C_ℓ computation method and the manual computation method of `compute_cls` predict the LJ Power Spectrum identically. This confirms the results of Figs. 1 and 2 that the underlying $a_{\ell m}$ sorting and C_ℓ computation methods work accurately. Then, we can see that the infinite shell computation creates a Power Spectrum of similar shape but different scaling compared to the predicted Power Spectrum, and the thin finite sphrical shell similar but with a different decay around multipole $\ell \sim 10^2$. This overplot is especially important because it culminates the work on the Power Spectra into one final plot, and represents the end for the Power Spectrum phase of the Bispectrum project.

4.2 Bispectrum Plots

The main portion of the Bispectrum, i.e. the Bispectrum calculations and plots, can be divided into three main sections. The first is the initial Last Journey (see [3] for information about Last Journey) gravitationally-induced weak-lensing Bispectrum plots that were computed from the filter average method of [7]. However, once we went to confirm that the Bispectrum calculations were correct using a theoretical $C_\ell \sim \ell^{-3}$ field, as explained in Section 3.2, we found some issues. The second section is devoted to the debugging and fine-tuning of the theoretical Bispectrum calculations on the $C_\ell \sim \ell^{-3}$ field. The third section would be devoted to the Convergence Bispectrum, but those plots are still currently being worked on.

4.2.1 Last Journey Bispectrum

These figures are the initial gravitationally-induced weak-lensing Bispectrum figures on the Last Journey density map for equilateral ℓ triplets (see [3]). They were computed by taking the even multipole ℓ triplets (as the odd ones are just 0) in linear space and in log space. In both figures, i.e. Figs. 4 and 5, there is a lot of noise in the resultant Bispectrum. These plots were the initial, incorrectly-scaled versions of the Bispectrum computation and using the `spherical` Wigner-3j function, which, as we know from Section 2.3, was prone to errors. These plots are still important to show because they represent the initial test run of the filter average method of computing the Bispectrum from Section 2.1.1. Furthermore, they gave an underlying Bispectrum plotting function which I was able to use for testing later. An important note about this is that, since the Bispectrum can have multiple kinds of shapes that are not just the equilateral ones depicted here, such as squeezed and orthogonal, this poses a challenge for plotting — hence why these plots are all for equilateral ℓ triplets (in addition to the fact that testing equilateral Bispectrum is more direct).

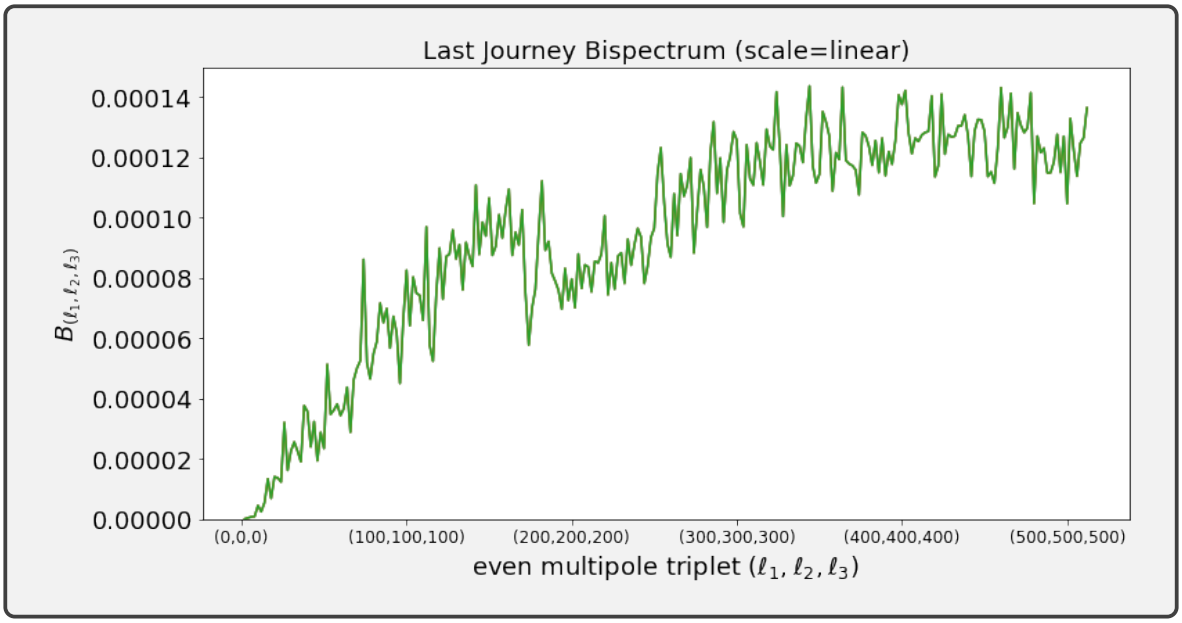

Figure 4: The figure above shows the Last Journey Bispectrum plotted on a linear scale for equilateral even ℓ triplets. As can be seen, there is a lot of noise in the actual computation.

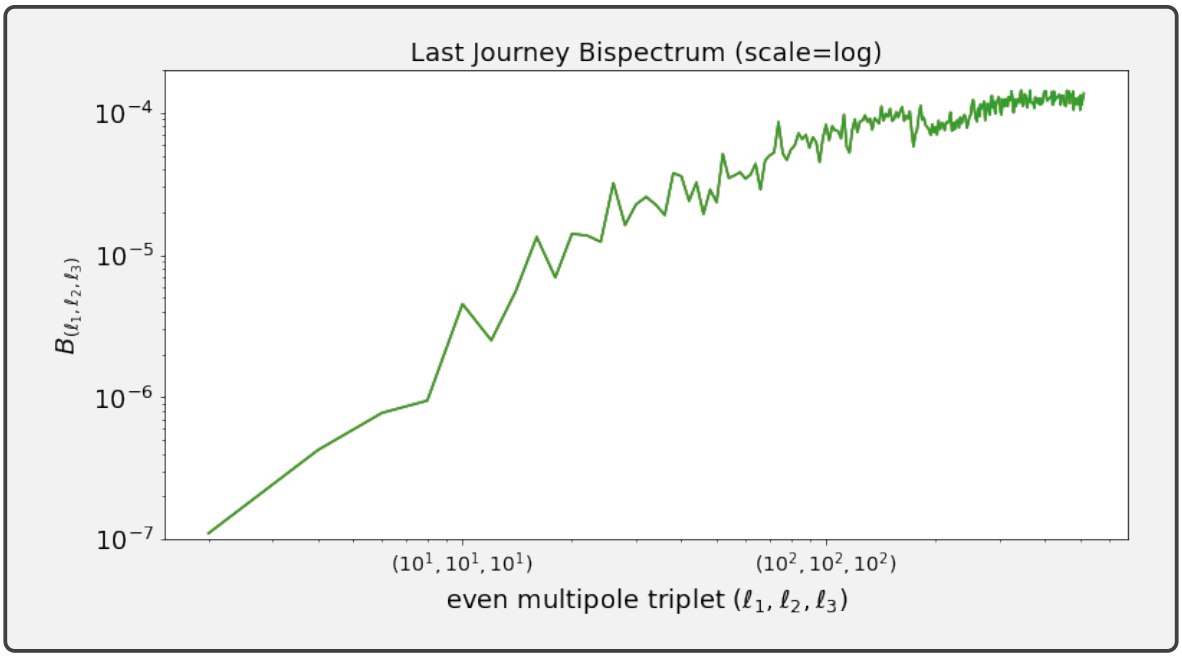

Figure 5: The figure above shows the Last Journey Bispectrum plotted on a log scale for equilateral even ℓ triplets. Similar to Fig. 4, there is a lot of noise in the Bispectrum, particularly around multiple triplet $\ell \sim 10^1$.

4.2.2 Theory Bispectrum Comparison

The figures show key results from when we were testing the filter average method of computing the Bispectrum on a theoretical density field $C_\ell \sim \ell^{-3}$. As explained in Section 3.2, during this testing, we discovered some issues in the scaling of the code, and these plots were instrumental in fixing those issues.

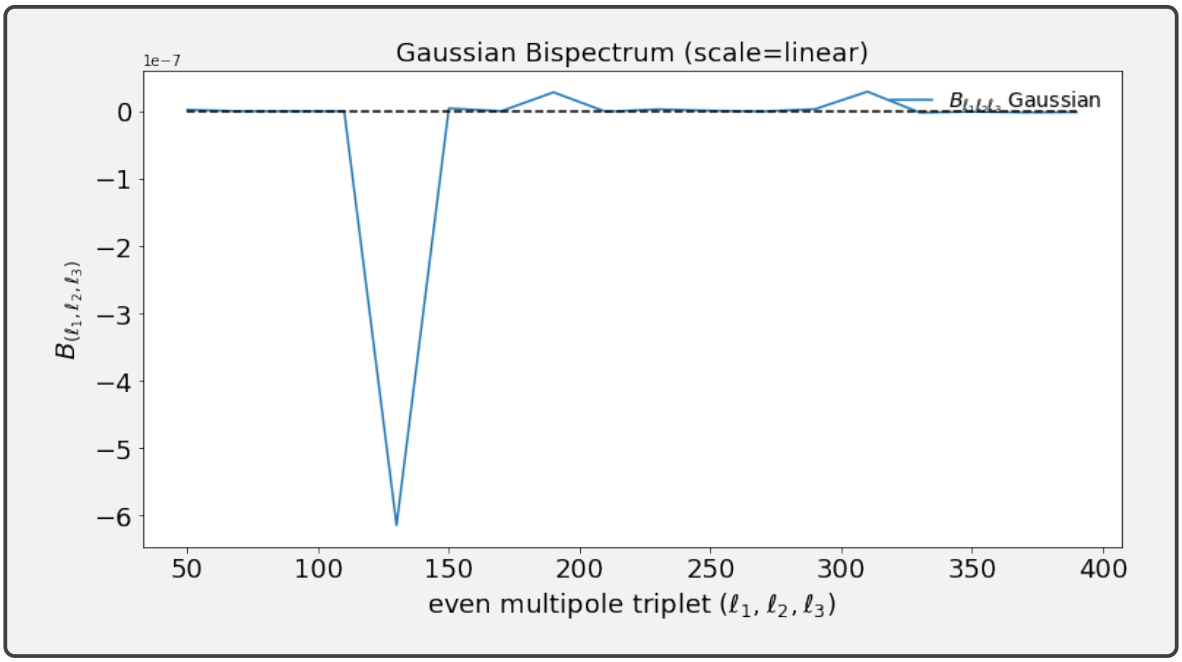

Figure 6: The figure above illustrates the Gaussian gravitationally-induced weak-lensing Bispectrum on a linear scale for equilateral even multiple triplets (ℓ_1, ℓ_2, ℓ_3) using the filter average method and the `pywigxjpf` Wigner-3j functions. There is one large spike around $\ell \sim 125$, with some other small noise spikes around $\ell \sim 195$ and $\ell \sim 315$. However, these are on the order of 10^{-7}, indicating that these spikes are likely just background noise. The Bispectrum we computed should theoretically be very close to 0 because it is Gaussian (and the Bispectrum is a measure of non-Gaussianity, see Section 1), implying that any noise in the resultant computation would be exacerbated when plotting. The Bispectrum computed for a non-Gaussian map, such as Last Journey, we would likely see a lot distribution on an order-of-magnitude a lot higher than 10^{-7}, trumping the small noise variation we see above. During testing, the scaling of the noise seen above was a lot larger, on the order of 10^{-2}, indicating that there was a problem with the Gaussian Bispectrum computation. After fixing some missing factors and method-testing our Wigner-3j calculations, as described in Section 2.3, we found the key finalized plot above for the theoretical ℓ^{-3} field.

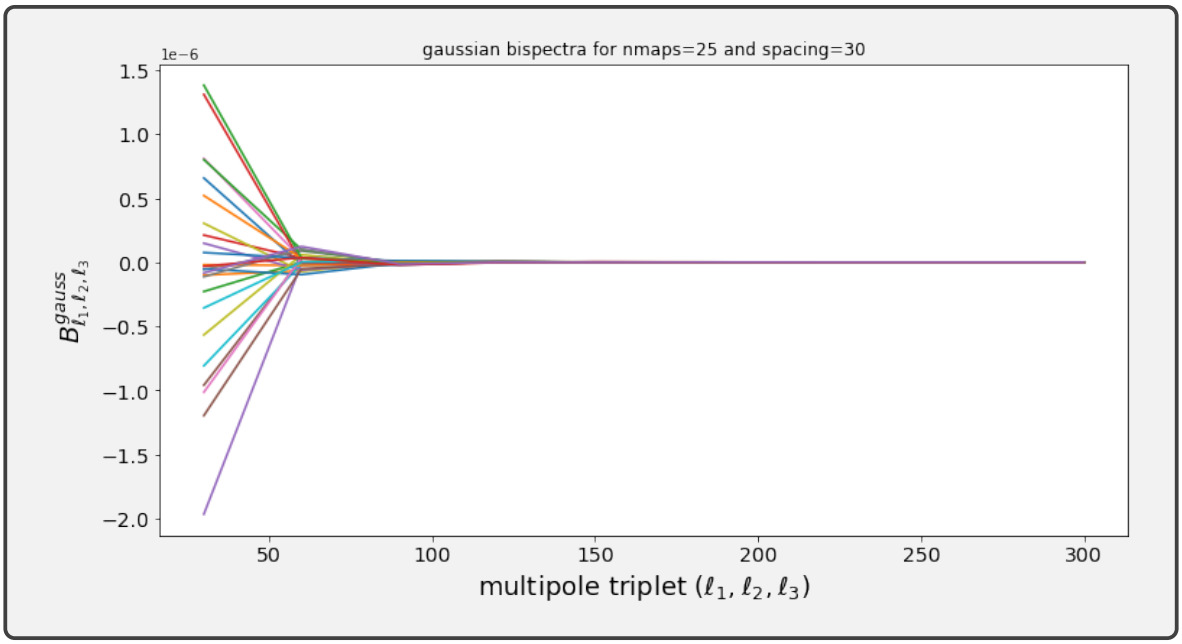

Figure 7: The figure above highlighs the Gaussian Bispectra as computed for 25 maps, with intervals of $\ell = 30$ between Gaussian Bispectra computation for ℓ triplets in the range of $\ell \in [0, 300]$. Each line color represents the Gaussian Bispectra computed for each different map. We can see that there is a lot of spread in the initial computation of the Bispectrum up to around $\ell \sim 60$, but by $\ell \sim 85$, they all seem to even out to around 0. Importantly, the spread is all around a range of $B_{\ell_1\ell_2\ell_3} \in [-2.0 \cdot 10^{-6}, 1.5 \cdot 10^{-6}]$ — a relatively low order-of-magnitude range. As explained above, since the Gaussian Bispectrum should be close to 0, the low values of this Bispectrum was a good indicator to us that we are probably just seeing some noise from the $C_\ell \sim \ell^{-3}$ field that will be trumped by a Bispectrum computation on an actual non-Gaussian density field. The converging of all of the Gaussian Bispectra of the 25 maps to 0 after $\ell \sim 85$, is what we would expect, which is also a positive indicator.

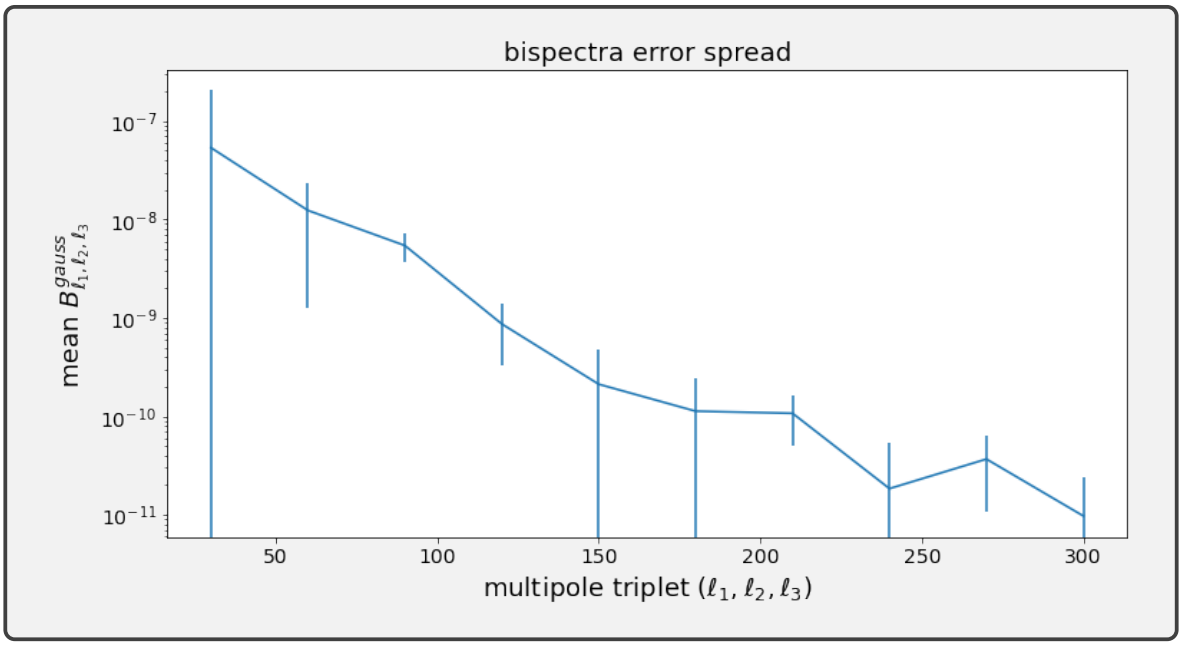

Figure 8: The figure above is a continuation of Fig. 7, as it highlights the error spread of the mean Gaussian bispectra for the maps above. It can be observed from the plot that our error spread is pretty small, on the order of $B^{\text{gauss}}_{\ell_1\ell_2\ell_3} \sim 10^{-7}$ to $B^{\text{gauss}}_{\ell_1\ell_2\ell_3} \sim 10^{-11}$.

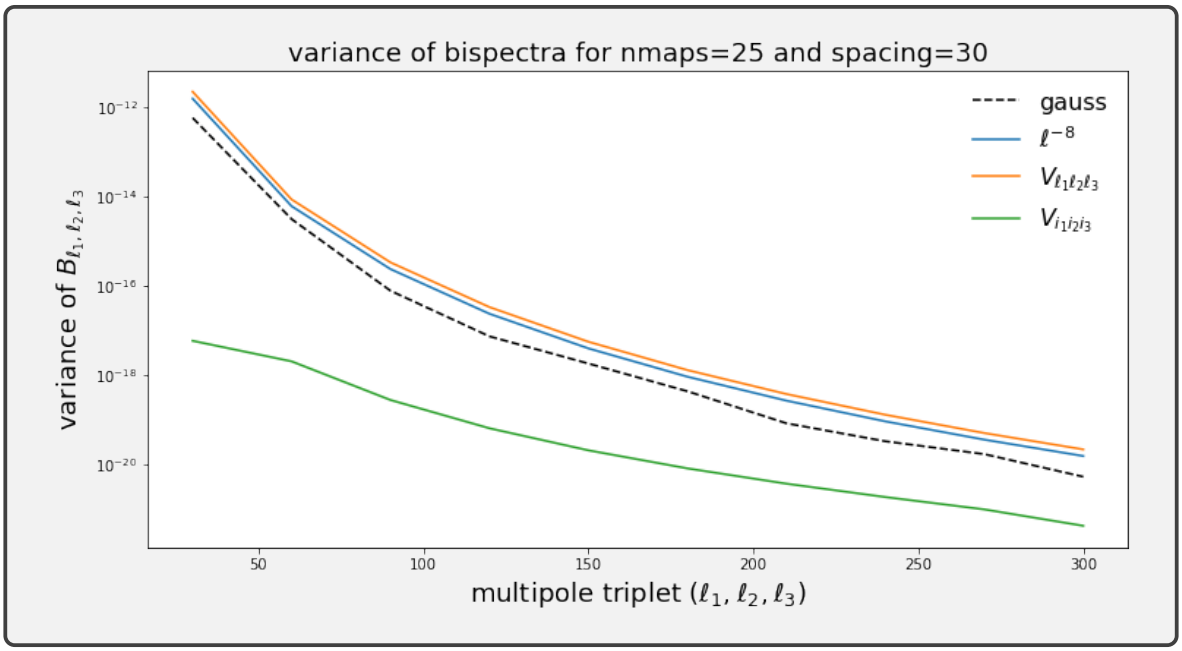

Figure 9: The figure above demonstrates the computed variances for the Gaussian Bispectra of the 25 maps from Figs. 7 and 8. The black dashed line represents the Gaussian variance, as computed from [7]. The solid blue line represents the theoretical ℓ^{-8} variance of a $C_\ell \sim \ell^{-3}$ field. The solid orange line represents the computed $V_{\ell_1\ell_2\ell_3}$ variance that is to be expected from the filter average method of Bispectrum calculation, as outlined in Section 2.1.1. Finally, the solid green line represents the computed $V_{i_1i_2i_3}$ variance that is to be expected from the *binned* Bispectrum estimator for the filter average method of Bispectrum calculation, as also outlined in Section 2.1.1. From the plot, we can observe that the predicted variance $V_{\ell_1\ell_2\ell_3}$ follows closely with a variance of ℓ^{-8} that is expected for the theoretical field we used, and is very close to the Gaussian variance as well. This is important because the reason for using a theoretical field in the first place was to confirm the accuracy of our Bispectrum calculations and resulting variance calculations from [7] against something that we can directly compare it against, i.e. the $C_\ell \sim \ell^{-3}$ field. Since all three of these variances follow each other closely, we were able to determine that the Gaussian Bispectrum calculations from [7] were likely computed mostly correctly. The solid green $V_{i_1i_2i_3}$ is there for the binned case of the Gaussian Bispectrum as a reference point. Obviously, it does not follow the non-binned estimator case, as expected. A future test would be to go back and plot the variance of our now fully-functioning binned Bispectrum estimator against this expected variance $V_{i_1i_2i_3}$.

Overall, what we observed from these gravitationally-induced weak-lensing Bispectrum computations on a theoretical field is that we were on the right track, albeit with some scaling issues here and there. The plot infrastructure was also good for testing, and proved extremely useful for the Map method of Bispectrum computation in Section 2.1.2. As explained in Section 3.2, once we fixed the scaling issue in the filter average method with the help of the map method code, we were able to finalize a Bispectrum calculator and compile it all into `bispec_calculator.py`. Using this updated and hopefully correct gravitationally-induced weak-lensing Bispectrum calculator on the Last Journey density maps would be a good direction for future research.

4.2.3 Convergence Bispectrum

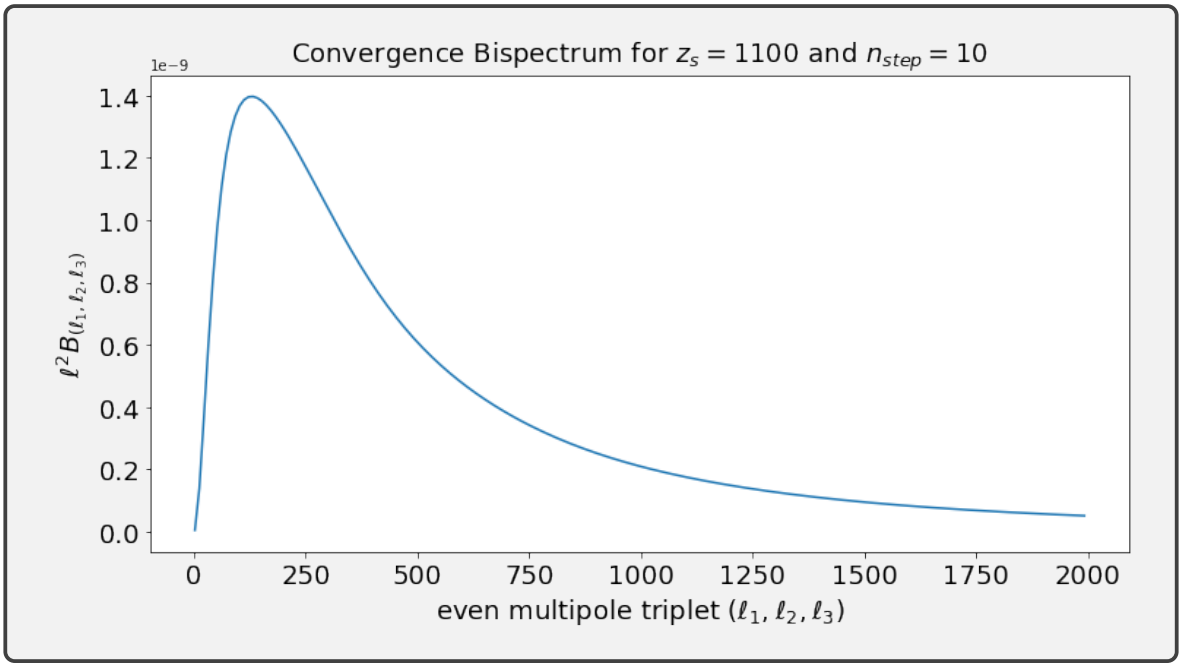

Figure 10: The figure above depicts the Convergence Bispectrum for the CMB at redshift $z_s = 1100$ for even equilateral multiple triplets $(\ell_1\ell_2\ell_3)$ with $\ell_{max} = 2000$ on a linear scale. The underlying cosmology is based on Planck 2018 parameters. The shape follows a very similar structure to that expected for equilateral Bispectra with these parameters from [8], but note that there is still a present scaling issue that is being fixed.

5 Conclusion

5.1 Summary

Overall, this report summarizes code responsible for computing the gravitationally-induced weak-lensing Bispectrum via two different methods from [7] — namely the filter average method and the map method — along with the Convergence Bispectrum from [8]. Information about this code, and the Github Repository that houses it, can be found in Section 3. This report further compiles much of the theory and supporting papers necessary to create this code in Section 2.1. Section 2 also includes pertinent information about the methodology and techniques used to calculate the Bispectrum using Numba and Wigner-3j calculators, along with issues that arose during this process, in Sections 2.2 and 2.3. Relevant figures that were found using this code can be found in Section 4 as they were applied to Last Journey data (see [3]) and theoretical data. Information about the Bispectrum and the project itself can be found in Section 1. Directions for Future Research in the weak-lensing Bispectrum can be found in the next section (Section 5.2). Concluding Thoughts and Acknowledgements can be found in Sections 5.3 and 5.4 respectively.

5.2 Directions for Future Research

There are four main directions for future research, which I will classify into two short term and two long term tasks.

The first short term task to work on includes fixing up a few of the scaling issues we had with the convergence bispectrum calculations, which we are currently in the process of doing, and then making sure that they agree with some of our previous plots. There were a few missing h factors and unitary c factors in the initial write-up for the Convergence Bispectrum code that caused the power spectrum $P(k)$ to be slightly off by a few orders of magnitude. Fixing these issues should allow us to find a convergence bispectrum in agreement with previous calculations from [8].

The second short term task would be to confirm and package up the full Bispectrum Calculator, then make sure it works with Argonne HACC simulations. This way, we can test different initial conditions and non-Gaussianities on the Bispectrum calculator and see how they stack up. The first step in doing this would be to actually test the gravitationally-induced weak-lensing Bispectrum calculator in `bispec_calculator.py` on a few Last Journey density maps, and possibly again on the theoretical $C_\ell \sim \ell^{-3}$ field, to confirm accuracy. Once we confirm this and package things up, then we can move on to integrating with Argonne HACC simulations.

For the first long term task, looking into the effects of masking would be an important direction for future research. Masking refers to the process of flagging or excluding certain areas of the map where data might be missing or contaminated by foreground sources. These masked regions can introduce complications when calculating statistical measures like the bispectrum. This is important because our maps might not always be of the full sky, either because of contamination or just because we are looking at a smaller portion of the sky Investigation these impacts and how can account for masking would be very important. Two possible methods for doing so that we discussed could include masking-aware estimators and mask inpainting. The former refers to weighing masked pixels differently during bispectrum computation, and the latter refers to filling in the missing or masked regions of the sky maps using interpolation or other statistical methods.

For the second long term task, since we have a binned version of the bispectrum estimator working, a direction we could take is to look into extending the theory for the weak lensing bispectrum into a binned version and seeing how it differs.

5.3 Final Thoughts

Overall, this project was an exciting and engaging way to learn more about Cosmology and Scientific Computing. I learned a lot about the Power Spectrum, Bispectrum, the CMB, Python, High-Performance Computing, and Computational Cosmology overall, for which I am very grateful and eager to apply in the future. While daunting at times, I am grateful for the opportunity to explore my love of the cosmos through the Bispectrum and Computational Cosmology. This project has a lot more that can be explored, and I can't wait to see where it goes.

5.4 Acknowledgements

Thank you to Dr. Patricia Larsen for being my mentor and guiding me through this Bispectrum project — whose time, dedication, and thoughtfulness was instrumental, and without whom this

work would not be where it is today. Thank you to Dr. Salman Habib for giving me this opportunity at Argonne to work on this project and helped me fix the Wigner-3j issues. Thank you to Dr. Michael Buehlmann and Dr. JD Emberson for helping me understand and implement parallelization techniques within my code, and for helping me better understand the Power Spectrum and Bispectrum. A very big thank you to Aurora Cossairt, who always took the time to help me understanding the intricacies of the Power Spectrum and the Bispectrum and the computations behind them. Thank you to Humza Qureshi, who helped advance my Python knowledge as I began working on this project.

6 Bibliography

References

[1] Martin Bucher, Benjamin Racine, and Bartjan van Tent, *The binned bispectrum estimator: template-based and non-parametric cmb non-gaussianity searches* (2016), available at 1509.08107.

[2] Nora Elisa Chisari, David Alonso, Elisabeth Krause, C. Daniellle Leonard, Philip Bull, Jérémy Neveu, Antonia Sierra Villarreal, Sukhdeep Singh, Thomas McClintock, John Ellison, Zilong Du, Joe Zuntz, Alexander Mead, Shahab Joudaki, Christiane S. Lorenz, Tilman Troester, Javier Sanchez, Francois Lanusse, Mustapha Ishak, Renée Hlozek, Jonathan Blazek, Jean-Eric Campagne, Husni Almoubayyed, Tim Eifler, Matthew Kirby, David Kirkby, Stéphane Plaszczynski, Anze Slosar, Michal Vrastil, and Erika L. Wagoner, *CCL: Core Cosmology Library*, 2019.

[3] Katrin Heitmann, Nicholas Frontiere, Esteban Rangel, Patricia Larsen, Adrian Pope, Imran Sultan, Thomas Uram, Salman Habib, Hal Finkel, Danila Korytov, Eve Kovacs, Silvio Rizzi, Joe Insley, and Janet Y. K. Knowles, *The last journey. i. an extreme-scale simulation on the mira supercomputer*, The Astrophysical Journal Supplement Series **252** (2021jan), no. 2, 19.

[4] H. T. Johansson and C. Forssé n, *Fast and accurate evaluation of wigner 3j, 6j, and 9j symbols using prime factorization and multiword integer arithmetic*, SIAM Journal on Scientific Computing **38** (2016jan), no. 1, A376–A384.

[5] Antony Lewis and Anthony Challinor, *CAMB: Code for Anisotropies in the Microwave Background*, 2011.

[6] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz, *Sympy: symbolic computing in python*, PeerJ Computer Science **3** (January 2017), e103.

[7] D Munshi, T Namikawa, T D Kitching, J D McEwen, R Takahashi, F R Bouchet, A Taruya, and B Bose, *The weak lensing bispectrum induced by gravity*, Monthly Notices of the Royal Astronomical Society **493** (2020feb), no. 3, 3985–3995.

[8] Toshiya Namikawa, Benjamin Bose, François R. Bouchet, Ryuichi Takahashi, and Atsushi Taruya, *CMB lensing bispectrum: Assessing analytical predictions against full-sky lensing simulations*, Physical Review D **99** (2019mar), no. 6.

[9] Ryuichi Takahashi, Takashi Hamana, Masato Shirasaki, Toshiya Namikawa, Takahiro Nishimichi, Ken Osato, and Kosei Shiroyama, *Full-sky gravitational lensing simulation for large-area galaxy surveys and cosmic microwave background experiments*, The Astrophysical Journal **850** (2017nov), no. 1, 24.

[10] Nick Woods, *Cosmological predictions from the primordial universe: Non-gaussianities*, 2018.