`FnFast`: A Framework for Perturbative Calculations for Large Scale Structure

Jon Walsh

July 13, 2015

Standard perturbation theory (SPT) and its cousins, such has Lagrangian perturbation theory (LPT) and the effective theory of large scale structure (EFTofLSS) perform a similar set of basic computations: they compute tree and (multi-)loop diagrams, with nearly identical computational mechanics between theories. This package uses an abstract framework to represent and compute diagrams, with examples in both SPT and the EFTofLSS.

1 Installation

`FnFast` is written in C++ and uses C++11 features. It relies on the external packages `CUBA` (for integration) and `gsl` (for interpolation). The makefile is straightforward and only requires you to point to the locations of `CUBA` and `gsl` on your machine. Once you do this, a simple `make` will build the main programs.

2 Diagrams

The diagrams computed by SPT and similar theories fall into a very general category:

- Propagators (or edges) receive weights that are scalar functions of the momentum flowing in the line.
- Vertices receive weights that are scalar functions of all of the momenta flowing in/out of the vertex.
- In the simplest cases, the internal lines are all equivalent and symmetry factors that depend only on the topology of the graph may be used to count degenerate diagrams.
- Schemes for regularizing infrared (IR) poles in loop momenta can be algorithmically encoded rather easily.
- We can efficiently perform numeric integrals over the internal momenta (e.g. using Monte Carlo importance sampling methods such as VEGAS).

The goal of this package is to enable efficient implementation of these calculations, such that the work required in adding new diagrams (or making use of existing ones) is minimal.

`FnFast` represents diagrams in terms of the topology and the momentum flow in the propagators. Diagram objects exist independent of any particular values of the momenta; we only use 3-vectors (as represented in the code) when asking for the value of the diagram for given external/loop momenta. This is done by a labeling system:

$$\begin{aligned} \text{vertices: } & \{v_1, v_2, v_3, v_4\} \\ \text{momenta: } & \{q, k_1, k_2, k_3, k_4\} \end{aligned} \tag{1}$$

where q labels the loop momentum and the k_i label external momenta[1].

Using this labeling we can represent the vertex factors in a diagram through a simple map[2] from vertex labels to functions (such as the kernels in SPT). For example, for an SPT bispectrum diagram this map would be

```
// vertex functions
SPTkernels* kernels = new SPTkernels;
unordered_map<Vertices::VertexLabel, KernelBase*> vertex_kernels =
    {{Vertices::v1, kernels}, {Vertices::v2, kernels}, {Vertices::v3,
    kernels}};
```

Here, `SPTkernels` contain a member function `Fn` which calculates the value of the SPT kernels given a vector of momenta.

Similarly, lines are labeled by the vertices they go between and the momentum they carry. This is done in two steps. First, a `Propagator` object represents the momentum flowing in a line, in the form of a map from momentum labels to coefficients of that momentum. Second, a `Line` object carries a starting and ending vertex and a `Propagator` instance. For example, a line between vertices 1 and 2 with momentum $k_2 - q$ is declared:

```
Propagator prop_qk2(unordered_map<Momenta::MomentumLabel, double>
    {{Momenta::q, -1}, {Momenta::k2, 1}});
Line line_12(Vertices::v1, Vertices::v2, prop_qk2);
```

The linear power spectrum that comes along with each propagator can be simply called once we know the momentum flowing through the line.

We put these objects together to form a diagram. One can think of this as the `Line` objects specifying the topology of the diagram, and the additional objects (such as `vertex_kernels` above) specifying what functions to use for the vertex and propagator factors. As an example,

[1] For two loops and beyond, one can add additional labels; the same goes for vertices for $\geq$5-point functions.

[2] `FnFast` frequently uses the `unordered_map` data structure in `C++11` to connect labels to objects.

below is the code to declare the $B321b$ diagram:

$$\begin{aligned} v_1 &\to v_2 : q \\ v_1 &\to v_2 : k_2 - q \\ v_1 &\to v_3 : k_3 \end{aligned}$$

The code to construct this diagram is:

```
// B321b
// linear power spectrum
LinearPowerSpectrumCAMB PL_CAMB("data/PL_CAMB.dat");
// vertex functions
SPTkernels* kernels = new SPTkernels;
unordered_map<Vertices::VertexLabel, KernelBase*> vertex_kernels =
    {{Vertices::v1, kernels}, {Vertices::v2, kernels}, {Vertices::v3,
    kernels}};
// propagators
Propagator prop_q(unordered_map<Momenta::MomentumLabel, double>
    {{Momenta::q, 1}});
Propagator prop_qk2(unordered_map<Momenta::MomentumLabel, double>
    {{Momenta::q, -1}, {Momenta::k2, 1}});
Propagator prop_k3(unordered_map<Momenta::MomentumLabel, double>
    {{Momenta::k3, 1}});
// lines
Line line_12a(Vertices::v1, Vertices::v2, prop_q);
Line line_12b(Vertices::v1, Vertices::v2, prop_qk2);
Line line_13(Vertices::v1, Vertices::v3, prop_k3);
vector<Line> lines {line_12a, line_12b, line_13};
// define the diagram
Diagram* B321b = new Diagram(lines, vertex_kernels, &PL_CAMB);
```

Many of these objects may be common between diagrams, shrinking the code overhead per diagram. A good example of the time/space savings with this approach is that when the `Diagram` object is declared, combinatoric objects such as the symmetry factor and permutations of external momentum routings are automatically computed; these objects would be cumbersome if individual diagrams were explicitly encoded.

2.1 Conventions and Notes on Adding Diagrams

Although this approach is rather automated in terms of the construction of diagrams, it is not foolproof. Ideally, one could set up an algorithm to automatically generate diagrams in a given class or at a given order. In practice this becomes more complex as the number of types of vertices grows or loops are added, and comes with downsides in terms of accessibility of individual diagrams and the general structure of calculations. With a well-defined set of use cases, one can imagine an automated diagram generation framework being useful.

Therefore, when adding diagrams, one should follow a couple of conventions to avoid issues. First, use canonical labeling for vertices and momenta; e.g., for a 3-point diagram use `Vertices::v1, Vertices::v2, Vertices::v3` and `Momenta::k1, Momenta::k2, Momenta::k3`. Existing diagrams have a base momentum routing where the loop momentum is from vertex 1 to itself or vertex 1 to vertex 2.

Finally, the code assumes all external momenta are outgoing, so that for an n-point function one should use the relation

$$k_1 + k_2 + \ldots + k_n = 0\,.$$

3 Calculations and Main Programs

Groups of diagrams are bundled into calculations, such as the power spectrum, that `FnFast` provides a high-level interface for. These, along with main programs, should hopefully be all that typical users have to interact with. The average user should hopefully fall into one of two groups:

- Those making use of existing calculations, who write/modify `main` routines to perform a specific analysis (e.g., fitting c_s in the EFTofLSS).
- Those wanting to implement a new calculation and accompanying main program to perform an analysis (e.g., calculating the 2-loop bispectrum).

There is no interface or base class for calculations, since the user may want very different things from different calculations. The existing calculations are `PowerSpectrum`, `Bispectrum`, `Trispectrum`, all within the EFTofLSS (and therefore SPT as well). Each provides member functions to access the tree, loop, and counterterm diagrams at various levels of exclusivity (e.g., before and after loop integration for the loop diagrams).

4 To-Do

- Improved build system
- Design and implement a more parsimonious container to label objects according to momenta/vertices
- Factor out integration routines into common container
- Improve adaptive sampling for integration
- Build a `MathLink` interface
- Explore multi-threading options for integration through `CUBA`
- Implement 2-loop (and 3-loop?) functionality (with the power spectrum in mind)