An Introduction to `Hammer`: Helicity Amplitude Module for Matrix Element Reweighting

Stephan Duell,[1] Florian U. Bernlochner,[2] Zoltan Ligeti,[3] Michele Papucci,[3] and Dean J. Robinson[4,3]

[1]*Physikalisches Institut der Rheinischen Friedrich-Wilhelms-Universität Bonn, 53115 Bonn, Germany*

[2]*Karlsruher Institute of Technology, 76131 Karlsruhe, Germany*

[3]*Ernest Orlando Lawrence Berkeley National Laboratory, University of California, Berkeley, CA 94720, USA*

[4]*Santa Cruz Institute for Particle Physics and Department of Physics, University of California Santa Cruz, Santa Cruz, CA 95064, USA*

Abstract

The manual...

CONTENTS

I. INTRODUCTION

The persistent $R(D^{(*)})$ results may be one of the first signals of new physics. Properly understanding their origin is imperative, which in turn requires the ability to produce fully simulated Monte-Carlo datasets for arbitrary new physics (NP) or hadronic models. The software tools to do this efficiently and self-consistently – a requirement to establish these anomalies as a NP signal – do not currently exist.

The extraction of $B \to D^{(*,**)}\tau\nu$ data from experimental analyses is complicated by multiple confounding factors: Neither the τ nor $D^{(*,**)}$ are visible final states in the B decays, instead decaying promptly in the detector. The actual final states involve missing energy in the form of at least two neutrinos, and the non-trivial masses of the τ and D^* lead to large interference effects in the differential decay distributions, formally of order m_τ/m_B in the SM and $\mathcal{O}(1)$ with NP included [1]. Combined with detector-level phase space cuts and down-feed contributions from D^{**} excited states, that can themselves exhibit large sensitivity to NP [2], these effects together imply that $B \to D^{(*,**)}\tau\nu$ signal data is itself *model-dependent*, both in the BSM and hadronic senses. Proper analysis of 'best-fits' in the space of NP theories is thus better achieved with *forward-folding* of all NP and hadronic effects, rather than attempts to unfold data into a simpler $B \to D^{(*,**)}\tau\nu$ process.

The `Hammer` software comprises a `C++` library designed to enable such forward-folded analysis by encoding the required efficient and self-consistent analysis of $b \to c\ell\nu$ decays for arbitrary new physics and hadronic models. This includes not only $B \to D^{(*,**)}\ell\nu$ decays ($\ell = \tau,\ \mu,\ e$) but other exclusive modes such as $\Lambda_b \to \Lambda_c\ell\nu$ or $B_c \to J/\psi\,\ell\nu$, among others. The design philosophy of `Hammer` is to construct tensor representations of amplitude-level results for the required high multiplicity final state decays – e.g. $B \to D^{(*,**)}(\to DY)\tau(\to X\nu)\nu$ – to permit computationally inexpensive exploration of NP or hadronic model effects in the fully differential phase space. An efficient event reweighting strategy, making use of these tensor forms, allows the user to perform studies on fully simulated Monte Carlo datasets – datatsets that incorporate e.g. phase space cuts, background and detector effects – as well as the effects of different schemes for form-factor parametrizations. Such studies may include fitting to the fully differential phase space of these processes, or histogramming in arbitrary dimensions.

While `Hammer` has been designed primarily with $b \to c\tau\nu$ processes in mind, the general framework has been designed to be extendable to processes such as $b \to s\ell\ell$, that also exhibit striking anomalies.

II. DESIGN OVERVIEW

A. Tensors and indices

The `Hammer` library is designed for the analysis of processes generated by theories that may be specified by a linear sum of operators $\mathcal{O}_\alpha$ and SM or new physics (NP) Wilson coefficients c_α, such that

$$\mathcal{L} = \sum_\alpha c_\alpha \mathcal{O}_\alpha \,. \tag{1}$$

We specify in Sec V the conventions used for $B \to D^{(*,**)}\ell\nu$ processes. This linear structure is preserved in the amplitudes relevant for processes of interest, which may also be further linearized in a basis of form factors (FF), F_i, or structure functions that encode the physics of hadronic transitions. In general, then, an amplitude may be written in the tensorial form

$$\mathcal{M}^{\{s\}}\big(\{q\}\big) = \sum_{\alpha,i} c_\alpha \, F_i\big(\{q\}\big)\, \mathcal{A}_{\alpha i}^{\{s\}}\big(\{q\}\big)\,, \tag{2}$$

in which $\{s\}$ are a set of external quantum numbers and $\{q\}$ the set of external four-momenta. The object $\mathcal{A}_{\alpha i}$ is an NP- and FF-generalized *amplitude tensor*. In the case of cascades, relevant for $B \to D^{(*,**)}(\to DY)\tau(\to X\nu)\nu$ decays, the amplitude tensor may contain coherent sums over several sets of internal quantum numbers (see Sec V for τ spinor and $D^{(*,**)}$ polarization phase conventions). In all processes currently handled by `Hammer` (see Sec. III B for a list), the $b \to c$ form factors are exclusively functions of

$$q^2 = \big(p_B - p_{D^{(*,**)}}\big)^2\,, \tag{3}$$

or equivalently functions of dimensionless kinematic variable w,

$$w = v \cdot v' = \frac{m_B^2 + m_{D^{(*,**)}}^2 - q^2}{2 m_B m_{D^{(*,**)}}}\,, \tag{4}$$

with $v = p_B/m_B$ and $v' = p_{D^{(*,**)}}/m_{D^{(*,**)}}$ the four-velocities of the initial and final states. The $\tau \to n\pi$, $n \geq 3$ structure functions are dependent on multiple invariant masses of the final state pions, so that processes containing such hadronic τ decays involve at least two sets separate sets of hadronic functions at amplitude level.

The corresponding polarized differential rate

$$d\Gamma^{\{s\}} = \sum_{\alpha,i,\beta,j} c_\alpha c_\beta^\dagger \, F_i F_j^\dagger\big(q^2\big)\, \mathcal{A}_{\alpha i}^{\{s\}} \mathcal{A}_{\beta j}^{\dagger\{s\}}\big(\{q\}\big)\, d\mathcal{PS}\,, \tag{5}$$

so that the outer product of the amplitude tensor, $\mathcal{W} = \mathcal{A}\mathcal{A}^\dagger$, is a *weight tensor.* (The phase space differential form includes on-shell δ-functions and geometric or combinatoric factors, as appropriate.)

An FF parametrization is permitted to also carry 'error' eigenbasis or other indices. For instance, an FF parametrization with a parameter set $\{\mu\}$ can be linearized around a best-fit point, $\{\mu^0\}$, so that

$$F_i \mapsto F_{i,a}\big(q^2;\{\mu\}\big) = F_i\big(q^2,\{\mu^0\}\big) + F'_{i,a}\big(q^2,\{\mu^0\}\big)\,, \tag{6}$$

where $F'_{i,a}$ is the perturbation of F_i in the ath principal component of the parametric fit correlation matrix. Alternatively, one can contemplate FF parametrizations that are linearized with respect to a basis of parameters, with index a. The key point here is that objects contracting on the a index are q^2 independent, and factor out of any phase space integral.

The calculational core of `Hammer` computes the amplitude tensor $\mathcal{A}$ event-by-event for any process having an corresponding extant amplitude class (see Sec. III B for a list), and as specified by initialization choices (more detail is provided below). Together with a specified form factor parametrization, F_i, and NP Wilson coefficients, c_α, an event NP weight can be rapidly computed by a linear contraction of the amplitude tensor (or, where relevant, weight tensor) with the NP and FF 'vectors', c_α and F_i.

B. Reweighting

Reweighting an event sample with weights w_i from an 'old' to a 'new' point in theory space (or to a different FF model) requires, at a minimum, the computation of the ratio

$$r_i = \frac{d\Gamma_i^{\text{new}}/d\mathcal{PS}}{d\Gamma_i^{\text{old}}/d\mathcal{PS}}\,, \tag{7}$$

applied event-by-event via the mapping $w_i \mapsto r_i w_i$. The 'old' or 'denominator' theory is typically chosen to be the SM plus a FF parametrization, and/or may be composed of pure phase space (PS) elements .

In certain use cases, it may be useful to compute and fold in (the SM component of) an overall ratio of rates $\Gamma^{\text{old}}/\Gamma^{\text{new}}$. Furthermore, in some cases the V_{cb}-normalized rate itself, $\Gamma^{\text{new}}/|V_{cb}|^2$, may be required. For example, if the MC sample has been initially generated with a fixed overall branching ratio, $\mathcal{B}_{\text{input}}$, one might wish to 'undo' this constraint via an additional factor $\mathcal{B}_{\text{new}}/\mathcal{B}_{\text{input}}$.

These various different components are computed by `Hammer` in the most general possible tensorial form:

(i) An NP- and FF-generalized tensor for $d\Gamma_i^{\text{new}}/d\mathcal{PS}$ is computed via the weight tensor (5), event-by-event, for all specified processes. An FF-generalized tensor is similarly computed for $d\Gamma_i^{\text{old}}/d\mathcal{PS}$, or just the overall normalization for the case that the old theory is pure phase space. The ratio r_i is then itself generally at least a rank-4 tensor.

(ii) The overall $b \to c\ell\nu$ rates $\Gamma^{\text{old, new}}$ need be computed only once for an entire sample. Hence, even if only SM components might be required in practice, it is computationally inexpensive to calculate the full NP-generalized tensor structure for each rate. However, it should be noted that these rates require integration over the $c\ell\nu$ Dalitz space: Since $b \to c$ form factors are q^2 dependent, the FF parameterization schemes for 'old' and 'new' must be specified, computed and contracted into the $b \to c\ell\nu$ weight tensors *before* integration. Depending on the FF parametrization, additional FF 'error' indices may still be present (see eq. (6)), so that $\Gamma^{\text{old, new}}$ are nominally (two-index) NP-generalized *rate tensors*, but may also feature two or more FF error indices too.

In the examples supplied with the source code, we show various different reweighting implementations and use cases, that make use of these objects in various different combinations.

C. Primary Functionalities

The user may choose when or whether the reweighting tensors $r_i w_i$ should be contracted into numerical weights for a specific theory or FF point, or alternatively stored for later reloading or binned into histograms. With regard to histogramming, it should be noted that contraction of the NP (or FF error, q^2 independent) indices of a set of $r_i w_i$ (re)weighting tensors commutes with binning by kinematic observables. This generates *NP-generalized histograms*, whose entries can be rapidly computed for any NP theory of interest. However, if such binning implicitly integrates over q^2 – q^2 is not an explicit dimension of the histogram – then necessarily contraction over the q^2 dependent FF indices is required first, before binning. At present, `Hammer` enforces contraction with FF indices before binning, for all variables.

The architecture of `Hammer` is designed around several primary functionalities:

(i) Receive instructions for which processes are 'included' to be reweighed, and which (possibly multiple schemes for) form factor parametrizations are to be used.

(ii) Read the included events from the event sample, and compute their corresponding NP- and FF-generalized amplitude or weight tensor, as well as the respective rate tensors, as needed.

(iii) Bin the event weight tensor – i.e. $r_i w_i$, as in eq. (7) – into histograms, as instructed.

(iv) Contract generalized weight tensors or bin entries against specific FF schemes or NP choices, to generate an event or bin weight.

(v) Save or reload amplitude or weight tensors or generalized histograms.

The schematic architecture and logical flow of `Hammer`, used to implement these functionalities, is shown in Fig. 1. Examples of the implementation of these functionalities is shown in the examples below.

Of notable importance is the use of two input cards: an initialization card and a parameter card. The initialization card is used, for programmatic reasons, to specify:

- Which decays should or should not be processed by `Hammer`. This specification is interpreted by a parser, that finds all intersections of the quoted (sub)processes with processes available to `Hammer`. The inclusion or exclusion of processes may be achieved by explicitly listing the decay cascades, or by listing just final states or subprocesses of interest.
- The input, aka 'old' aka 'denominator', FF parametrization.
- The output, aka 'new' aka 'numerator', FF parametrization(s): For the purpose of histogramming or weight storage, multiple numerator FF parametrizations may be specified in uniquely named schemes.

The parameter card lists the values of FF parameters for the FF parametrizations of interest, as well as values of Wilson coefficients to be contracted with the reweighting tensors. Examples of the use of both cards are shown below. The functionality of both cards may also be implemented programmatically, if the user wishes.

D. Code flow

A `Hammer` program may have two different types of structure: An *initialization* program, so called as it runs on MC as input, and may generate hammer format files; or a *reanalysis* program, which may reprocess histograms or event weights that have already been saved in an initialization run.

An initialization program has the generic flow:

(i) Create a `Hammer` object.

(ii) Declare included or forbidden processes, via `includeDecay` and `forbidDecay`.

(iii) Declare form factor schemes, via `addFFScheme` and `setFFInputScheme`.

(iv) (Optional) Add histograms, via `addHistogram`.

(v) (Optional) Declare the MC units, via `setUnits`.

(vi) Initialize the `Hammer` class members with `initRun`.

(vii) (Optional) Change FF default settings with `setOptions`, or (if not SM) declare the WCs for the input MC via `setWilsonCoefficients`.

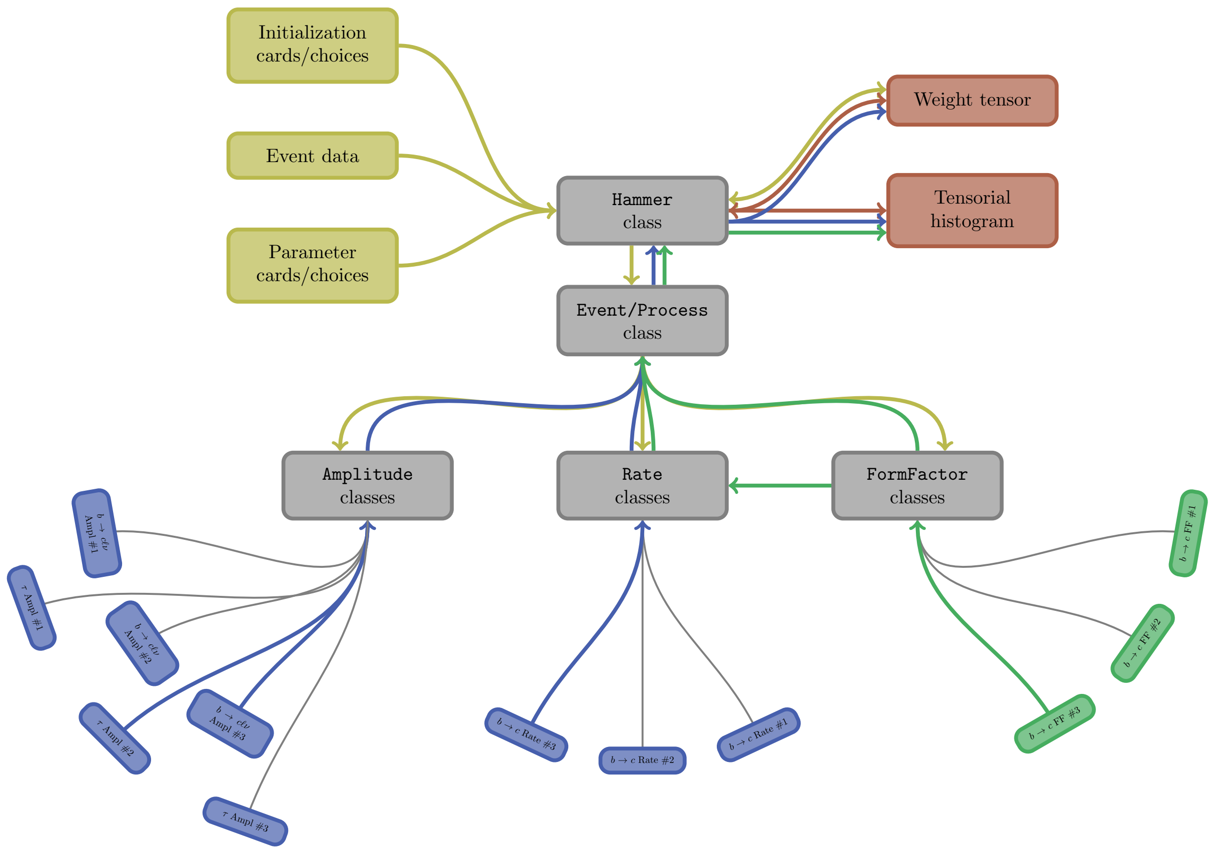


FIG. 1. The 'flying spaghetti monster' schematic architecture of `Hammer`. The flow of user specified choices or event data is shown by yellow arrows. Blue (green) arrows denote the flow of calculational information, in particular amplitude, weight or rate (form factor) tensors for the process specified in the event data. Red arrows highlight the flow of `Hammer` output, which may be saved or reloaded. Most internal `Hammer` classes are not shown in this schematic.

(viii) Each event may contain multiple processes, e.g. a signal and tag B decay. Looping over the events:

  (a) Initialize event with `initEvent`. For each process in the event:

    i. Create a `Hammer Process` object.

    ii. Add particles and decay vertices to create a process tree, via `addParticle` and `addVertex`.

    iii. Decide whether to include or exclude processes from an event via `addProcess` and/or `removeProcess`.

  (b) Compute or obtain event observables – specific particles can be extracted with `getParticlesByVertex` or other promgrammatic means – and specify the corresponding histogram bins to be filled via `fillEventHistogram`. (Alternatively, `setEventHistogramBin` allows one to instead specify the bin index directly).

   (c) Initialize and compute the process amplitudes and weight tensors for included processes in the event, and fill histograms with event weights – the direct product of include process weights – via `processEvent`.
   (d) (Optional) Save the weight tensors for each event, with `saveEventWeights`.

(ix) (Optional) Generate histograms with `getHistogram(s)` and/or save them with `saveHistograms`. NP choices are implemented with `setWilsonCoefficients`, FF uncertainties are set with `setFFEigenvectors`.

By contrast, an *analysis* program (from a previously initialized sample, stored in a buffer) has the generic flow:

(i) Create a `Hammer` object and specify the input file.

(ii) Reinitialize containers – include or forbid specifications, FF schemes, or histograms – with `loadRunHeader` (after `initRun`). One may further declare additional histograms to be compiled (from saved event weight data) via `addHistogram`.

(iii) (Optional) Looping over the events:
   (a) Initialize event with `initEvent`.
   (b) If desired, remove processes from an event with `removeProcess`.
   (c) Reload event weights with `loadEventWeights`.
   (d) Specify histograms to be filled via `fillEventHistogram`.
   (e) Fill histograms with event weights via `processEvent`.

(iv) (Optional) Load saved histograms with `loadHistograms`, and/or generate histograms with `getHistogram(s)`. NP choices are implemented with `setWilsonCoefficients`.

III. THE HAMMER FORGE

In this section, we provide user-relevant details for the operation of the `Hammer` library. In particular, this section contains a more detailed explanation of information handling and rules enforced by the computational core in following user specifications, assembling amplitudes, or returning histograms or weights, in order to avoid (or explain) 'unexpected' behavior.

A. From the process tree to an amplitude tensor

Starting with a single 'head parent' particle, labelled by index 0 in Fig. 2, a typical decay encountered by `Hammer` can be represented in graphical terms as a 'process tree', in an obvious way: Each decay vertex is labelled by its local parent particle, connected to subsequent

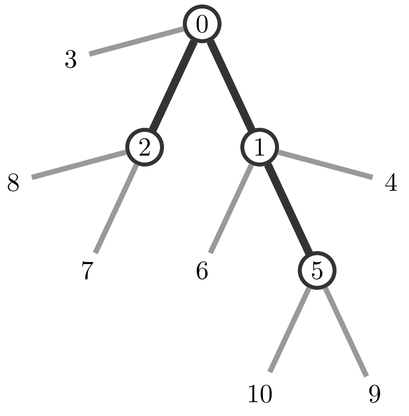


FIG. 2. Example process tree for a decay cascade involving 10 particles (numbers), 4 vertices (circles) and 3 edges (dark lines).

daughter decays by an edge (i.e. a line, or formally, a propagator). `Hammer` assembles the process tree through two methods `Process::addParticle` and `Process::addVertex`. The former adds a `Particle` class object – a momentum and a PDG code – to a container of particles; the latter fills a map of each parent to its daughters for each decay vertex.

From the filled process tree, `Hammer` determines several hashes or sets of hashes, that encode the structure of the tree: In particular, i) a set of the hashes of parent and daughter particle PDG codes at each vertex; ii) a combined hash for the process – a 'process ID' – providing a 1-1 identifier between the full decay cascade and a `size_t` integer. For any process, the latter can be obtained by the method `Process::getId`. The former will be relevant later for understanding how 'included' and 'forbidden' processes are identified.

At this stage, the natural computational step is to map each vertex into a corresponding amplitude tensor, contracting exchanged quantum numbers along each edge to form a single tensor for the whole process tree. In the simplest cases, this is precisely the strategy adopted by `Hammer`, i.e. the particle ID hashes constructed at each vertex are looked up in a dictionary of the signatures of available `Amplitude` classes. A similar technique, using the hash of the hadronic particles in a vertex, is used to identify whether form factors are needed at each vertex. (If form factors are required at a vertex, `Hammer` will obtain the relevant form factor parameterization as specified by the user for the hadronic transition in question.) If no amplitude is found for a vertex, hammer will simply skip this step of the cascade. This behavior means that hammer implicitly prunes potentially highly extended cascades, providing an amplitude tensor only for vertices `Hammer` 'knows' (i.e. the parts of the cascade we care about for understanding NP effects or FF parametrizations).

In certain cases the strategy adopted for determining the process amplitude is more sophisticated than a vertex-by-vertex approach. For certain decays, it can be computationally advantageous to calculate an amplitude for two adjacent amplitudes. For example, in $B \to (D^* \to D\gamma)\ell\nu$, simpler expressions can be obtained if one calculates the entire 'merged' amplitude, treating the D^* as an onshell internal state, rather than two separate amplitudes exchanging D^* spin. Similarly, for $\tau \to (\rho \to \pi\pi)\nu$, treatment of non-resonant effects from the broad ρ motivate expressing this amplitude as one merged amplitude, even though in the process tree it would be represented as two vertices. Multistep decays involving the broad D^{**} may also be more tractable when merged in this manner. Thus in additional to vertex amplitudes, `Hammer` is also capable of processing 'edge' amplitudes, that is, one amplitude belonging to two adjacent vertices connected by an edge in the process tree. It can therefore happen that although `Hammer` does not know the amplitude for a particular vertex, it does know an edge amplitude involving that vertex and another.

To explain what this means in practice for the user, it's useful to introduce a vertex and edge notation for the process tree. If `Hammer` knows the amplitude at a vertex, the vertex is denoted by a filled circle, and if unknown, by an open circle. If an edge vertex is available for two vertices, we connect them by a double line. This leads to five different types of amplitude combinations, defined in Table I. The arithmetic followed by `Hammer` in determining the amplitude from tree is as follows:

(i) Fill all available pure edges by lowest (i.e., furthest from head parent) to highest depth in the process tree, being sure not to assign the same vertex twice

(ii) Repeat for partial then full edges

(iii) Assign known vertex amplitudes to any remaining free vertices.

Two examples or this arithmetic are shown in Table II.

	Vertex		Edge		
Amplitude	Known	Unknown	Pure	Partial	Full
Notation	●	○	○=○	●=○	●=●

TABLE I. Definition of vertex and edge amplitude types.

B. Available vertex and edge amplitudes

The list of vertex and edge amplitudes known to `Hammer` are shown in Table III. Also shown are correspondingly available form factor parametrizations, as appropriate.

C. Including and excluding processes

The `Hammer` library contains an interpreter between a string representation of a vertex and the corresponding PDG codes of incoming and outgoing particles. At present, a string

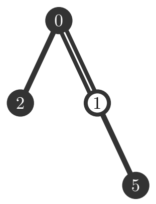
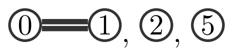
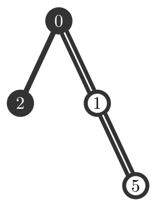
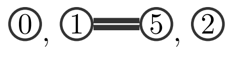
Known Amplitudes	Evaluated Amplitudes
	
	

TABLE II. Example arithmetic for filling amplitudes for the process tree of Fig. 2, assuming different example sets of known amplitudes in `Hammer`.

representation of a vertex, or 'vertex string' is formed by concatenating a single parent name with daughter names, in the form `ParentDaughter1Daughter2`.... The interpreter uses the syntax that particle names are parsed by a capital letter: the full list of names is provided in Table IV. The interpreter maps a vertex string to all possible *charge conserving* processes allowed by the charges of the specified particle names. For example the vertex string `D*DPi` is interpreted as all twelve possible $D^* \to D\pi$ vertices, while `D*+DPi` is interpreted as only the $D^{*+} \to D^+\pi^0$, $D^{*+} \to D^0\pi^+$, and (the heavily CKM suppressed) $D^{*+} \to \bar{D}^0\pi^+$ decays, and finally the vertex string `D*+D0Pi` corresponds to the unique decay $D^{*+} \to D^0\pi^+$.

The decay processes to be reweighed by `Hammer` are specified via `Hammer::includeDecay`, which takes a vector of vertex strings $\{V_1, V_2, \ldots, V_n\}$ as an argument, and may be invoked multiple times. Each `includeDecay` specification is *inclusive* and permits any process tree whose set of vertices P *contains* $\{V_1, V_2, \ldots, V_n\}$. The boolean logic applied by `includeDecay` is `AND` between each vertex string element, and `OR` between separate invocations of `includeDecay`. For example

```
ham.includeDecay({"BD*TauNu", "D*DGamma"});
ham.includeDecay({"BDMuNu"});
```

means 'Reweigh a process that either contains vertices ($B \to D^*\tau\nu$ **and** $D^* \to D\gamma$) **or** the vertex ($B \to D\mu\nu$)'. Hence e.g. $\bar{B}^0 \to (D^{*+} \to (D^+ \to K^+\pi^+\pi^-)\gamma)(\tau^- \to \ell^-\nu\nu)$ would be included. Radiative photons are automatically accounted for, and need not be specified in `includeDecay` specifications (see Sec. III M).

Processes are forbidden with the `Hammer::forbidDecay` method, which similarly takes a vector of vertex strings $\{V_1, V_2, \ldots, V_n\}$, and employs the same boolean structure as `includeDecay`. However, `forbidDecay` specifications are *exclusive* and forbids only process trees whose set of vertices P *equals* $\{V_1, V_2, \ldots, V_n\}$. For example

Process	Type	FF Parametrizations
$B \to D^{(*)}\ell\nu$	Vertex	`ISGW2`*, `BGL`*, `BGLVar`, `CLN`, `BLPR`
$B \to (D^* \to D\pi)\ell\nu$	Edge	`ISGW2`*, `BGL`*, `CLN`, `BLPR`
$B \to (D^* \to D\gamma)\ell\nu$	Edge	`ISGW2`*, `BGL`*, `CLN`, `BLPR`
$\tau \to \pi\nu$	Vertex	`---`
$\tau \to \ell\nu\nu$	Vertex	`---`
$\tau \to 3\pi\nu$	Vertex	`RCT`
$B \to D_0^*\ell\nu$	Vertex	`LLSW`, `BLR`
$B \to D_1^*\ell\nu$	Vertex	`LLSW`, `BLR`
$B \to D_1\ell\nu$	Vertex	`LLSW`, `BLR`
$B \to D_2^*\ell\nu$	Vertex	`LLSW`, `BLR`
$\Lambda_b \to \Lambda_c\ell\nu$	Vertex	`PCR`*, `BLRS`
Planned for next release		
$B \to (\rho \to \pi\pi)\ell\nu$	Edge	`BCL`*, `BSZ`
$B \to (\omega \to \pi\pi\pi)\ell\nu$	Edge	`BCL`*, `BSZ`
$B \to (J/\psi \to \ell\ell)\tau\nu$	Edge	
$\tau \to 4\pi\nu$	Vertex	
$\tau \to (\rho \to \pi\pi)\nu$	Edge	

TABLE III. Presently implemented amplitudes in `Hammer` and their types, and corresponding FF parametrizations or structure function tunes. SM only parametrizations are indicated by an asterisk.

```
    ham.forbidDecay({"B+D0barMuNu"});
```

means ‘Exclude a process that contains only the vertex $B^+ \to \bar{D}^0\mu^+\nu_\mu$’, but e.g. this would not exclude a process involving a subsequent D decay.

Inclusion or exclusion of processes may also be specified via an initialization card in YAML format. For example, the equivalent to the above `includeDecay` and `forbidDecay` invocations is

```
Include: [ [ BD*TauNu, D*Dpi ], BDMuNu ]
Forbid: [ B+D0barMuNu ]
```

Symbol	Particle(s)	Symbol	Particle(s)
D	D^+, D^-, D^0, $\bar{D}^0$	Lc+	Λ_c^+
D*	D^{*0}, D^{*-}, D^{*+}, $\bar{D}^{*0}$	Lc-	Λ_c^-
Lc	Λ_c^+ , Λ_c^-	Lb0	Λ_b^0
B	B^0, B^-, B^+, $\bar{B}^0$	Lb0bar	$\bar{\Lambda}_b^0$
Lb	Λ_b^0 , $\bar{\Lambda}_b^0$	Pi0	π^0
K	K^+, K^-, K_L^0, K_S^0	Pi+	π^+
Pi	π^0, π^+, π^-	Pi-	π^-
Tau	τ , τ^+	Nut	ν_τ
Nu	ν_e , $\bar{\nu}_e$, ν_μ , $\bar{\nu}_\mu$, ν_τ , $\bar{\nu}_\tau$	Nutbar	$\bar{\nu}_\tau$
Ell	μ^- , μ^+ , e^- , e^+	Num	ν_μ
D**0*	D_0^{*0}, D_0^{*-} , D_0^{*+} , $\bar{D}_0^{*0}$	Numbar	$\bar{\nu}_\mu$
D**1*	D_1^*, D_1^{*-} , D_1^{*+} , $\bar{D}_1^{*0}$	Nue	ν_e
D**1	D_1^0, D_1^- , D_1^+ , $\bar{D}_1^0$	Nuebar	$\bar{\nu}_e$
D**2*	D_2^{*0} D_2^{*-} , D_0^{*+} , $\bar{D}_0^{*0}$	W+	W^+
E	e^+ , e^-	W-	W^-
Mu	μ^- , μ^+	W	W^+ , W^-
Gamma	γ	D**0*0	D_0^{*0}
Tau+	τ^+	D**0*+	D_0^{*+}
Tau-	τ	D**0*-	D_0^{*-}
E+	e^+	D**0*0bar	$\bar{D}_0^{*0}$
E-	e^-	D**0*+-	D_0^{*+} , D_0^{*-}
Mu+	μ^+	D**0*abar	D_0^{*0} , $\bar{D}_0^{*0}$
Mu-	μ^-	D**1*0	D_1^{*0}
K+	K^+	D**1*+	D_1^{*+}
K-	K^-	D**1*-	D_1^{*-}
K0S	K_S^0	D**1*0bar	$\bar{D}_1^{*0}$
K0L	K_L^0	D**1*+-	D_1^{*+} , D_1^{*-}
B0	B^0	D**1*abar	D_1^{*0} , $\bar{D}_1^{*0}$
B+	B^+	D**10	D_1^0
B-	B^-	D**1+	D_1^+
B0bar	$\bar{B}^0$	D**1-	D_1^-
B+-	B^+ , B^-	D**10bar	$\bar{D}_1^0$
Babar	B^0 , $\bar{B}^0$	D**1+-	D_1^+ , D_1^-
D0	D^0	D**1abar	D_1^0 , $\bar{D}_1^0$
D+	D^+	D**2*0	D_2^{*0}
D-	D^-	D**2*+	D_2^{*+}
D0bar	$\bar{D}^0$	D**2*-	D_2^{*-}
D+-	D^+ , D^-	D**2*0bar	$\bar{D}_2^{*0}$
Dabar	D^0 , $\bar{D}^0$	D**2*+-	D_2^{*+} , D_2^{*-}
D*0	D^{*0}	D**2*abar	D_2^{*0} , $\bar{D}_2^{*0}$
D*+	D^{*+}		
D*-	D^{*-}		
D*0bar	$\bar{D}^{*0}$		
D*+-	D^{*+} , D^{*-}		
D*abar	D^{*0} , $\bar{D}^{*0}$		

TABLE IV. List of currently available particle specifications and corresponding particles.

using the same vertex string syntax and symbology.

D. Form factor schemes

In general, histogramming of event weights does not commute with contraction of FF parametrization and weight tensors (unless one of the histogram dimensions is explicitly q^2).

The `Hammer` library therefore allows the user to specify form factor schemes, or 'schemes' to be used in reweighting. A form factor scheme is a set of FF parameterization choices for each hadronic transition involving form factors (or structure functions), and is labelled by a 'scheme name'. These schemes are set by the method `Hammer::addFFScheme`, which takes a scheme name plus a map from hadronic string representation to FF parametrization. The hadronic string follows the same syntax and uses the same particle symbols as for vertex strings in Sec. III C. For example,

```
    ham.addFFScheme("Scheme1", {{"BD", "BLPR"}, {"BD*", "BLPR"}});
    ham.addFFScheme("Scheme2", {{"BD", "BGL"}, {"BD*", "CLN"}});
```

declares two different FF schemes, choosing BLPR for both $B \to D$ and $B \to D^*$ form factors in `"Scheme1"`, and a mixture of schemes for `"Scheme2"`. Separate histograms and event weights are generated for each scheme name, which are retrieved with the methods `Hammer::getHistogram(s)` and `Hammer::getWeight(s)`, as described below. The list of symbols for available FF parametrizations are provided in Table III. In principle, different FFs for charged and neutral processes can be set, e.g. via an entry `{"B+-D", "BLPR"}` versus `{"BabarD", "BLPR"}`, and so on.

Specification of the form factor schemes used to generate the MC sample, i.e. the denominator or input form factors, must be specified in order for `Hammer` to be able to generate the reweighting tensors. These schemes are specified by the method `Hammer::setFFInputScheme`, which takes a map from hadronic string representation to FF parametrization scheme. For example

```
    ham.setFFInputScheme({{"BD", "ISGW2"}, {"BD*", "ISGW2"}});
```

sets both $B \to D$ and $B \to D^*$ denominator form factors to ISGW2, a common MC parametrization.

As for the include and forbid specifications, the form factor schemes can also be specified in the initialization card in YAML format. The equivalent to the above settings is

```
FormFactors:
    NumeratorSchemes:
        Scheme1: { BD: BLPR, BD*: BLPR }
        Scheme2: { BD: BGL, BD*: CLN }
    Denominator: { BD: ISGW2, BD*: ISGW2 }
```

FF parametrization default settings are fixed inside the FF classes themselves. After invocation of `ham.initRun()`, manipulation of the FF default settings may be achieved via `setOptions`, which takes YAML format arguments. For instance,

```
    ham.setOptions("BtoDBGL: {ChiTmB2: 0.01, ChiL: 0.002}");
```

changes the two BGL outer function parameters from their default settings. (Note that the YAML key for the relevant FF class matches the format of the *class name*, with '`to`' inserted in the hadronic transition. E.g. `BtoDBGL`, rather than `BDBGL`.)

E. Form factor duplication

Duplication of the same FF class is permitted in different FF schemes, and is invoked by adding a token to a FF parametrization name, separated by an underscore. For instance, one may declare

```
ham.addFFScheme("Scheme1", {{"BD", "BGL_1"}, ... });
ham.addFFScheme("Scheme2", {{"BD", "BGL_2"}, ... });
ham.setFFInputScheme({{"BD", "BGL_den"}, ... });
```

In this case, three copies of the $B \to D$ BGL class are created, whose settings may be manipulated (via `setOptions`) separately. E.g.

```
ham.setOptions("BtoDBGL_1: {ChiT: 0.01, ChiL: 0.002}");
ham.setOptions("BtoDBGL_2: {ChiT: 0.03, ChiL: 0.007}");
ham.setOptions("BtoDBGL_den: {ChiT: 0.02, ChiL: 0.005}");
```

F. Units

While the reweights generated by `Hammer` are dimensionless, various form factor schemes are defined with respect to dimensionful quantities, requiring the library to know the units of the input MC. This is set by the `Hammer::setUnits` method, which accepts a string of the name of units convention, from `eV` to `TeV`. E.g. `ham.setUnits("MeV")`, declares the input MC to be in MeV. This declaration must be made before `initRun`.

The default units inside the library are GeV: The masses and partial widths in the `Pdg` are specified in GeV. These feed into rate computations, which are therefore also handled internally in GeV. (After events have been processed, `setUnits` may also be used to specify the units in which partial widths are returned by `getRate`. See Sec. III N below.)

G. Processing events

An `Event` object may contain multiple instances of `Process`, in order to account for the fact that a single event may feature e.g. two B decay processes. The `Event` class is initialized by `Hammer::initEvent()`, which may take an optional weight double if the event has a non-unit initial weight (this can also be set by `Hammer::setEventBaseWeight`). `Process` instances are added by `Hammer::addProcess(proc)` which also returns the `HashId` of the process. If the process is not allowed according to the `includeDecay` or `forbidDecay` specifications, the returned `HashId` is zero, and the process is not added to the relevant `Event` containers.

Once a process is added, it is automatically initialized, which chiefly involves: calculating the signatures of each vertex in the decay cascade; identifying the various subamplitudes making up the cascade, as well as relevant form factor parametrizations and vertex decay rates, for both the numerator/output and denominator/input. These amplitudes, form factors and rates are not computed, however, until the invocation of `Hammer::processEvent`. Once a process is added, the methods `Process::getParticlesByVertex` or `getVertexId` can be used to extract specific particles in a vertex or other vertex properties, taking as an argument the relevant vertex string. These methods can be used to construct desired observables belonging to the process; this can also be done by the user externally to `Hammer`, as desired. E.g.

```
proc.getParticlesByVertex("D*DPi");
```

returns `pair<Particle, vector<Particle>>` for the parent D^* and vector of daughter particles, D and π. As an additional convenience, a process can be explicitly removed from the event by `Hammer::removeProcess(procId)`, which takes the relevant process `HashId` as its argument. This functionality is mainly relevant if one wishes to use `Hammer`-supplied getter methods for extracting process observables, but one does not actually wish to include the process weight in computations. It can also be used to prevent inclusion of spurious processes in EventIds, that would otherwise cause the latter to undesirably proliferate in number.

Once all processes are added (and if histograms have been added, relevant ones have been filled; see Sec. III J), the amplitudes, weights and relevant rates are computed (and weights are added to histogram bins) by invocation of `Hammer::processEvent`. If `processEvent` is invoked on an event with no included (or all removed) processes, `Hammer` assigns a unit event weight to the event (times any initial weight specified in `initEvent` or `setEventBaseWeight`): Caution should therefore be employed in invoking `processEvent` on such events, if this behavior is not desired.

H. Retrieving event weights

Once an event has been processed (or loaded from a file), the weight for a specific event can be retrieved by `Hammer::getWeights("FFScheme")`, which returns a map of each process Id and corresponding `double` process weight for the specified FF scheme. These weights can then be combined as appropriate. Alternatively, if `HashId`s of the desired processes are known, one may use `Hammer::getWeight("FFScheme", procIds)`, where the second argument is a `vector<HashId>`, that returns the corresponding weights already combined into a `double`.

I. Setting Wilson Coefficients and FF eigenvectors

Crucial to the application of either `getWeight(s)` method is pre-setting of the relevant 'external data', i.e. WCs and FF uncertainties (if any). (Settings for FF parameter central values must be invoked before `Hammer::processEvent`, as must settings for the denominator/input WCs; see Sec. III G.) The WCs are set by the method `Hammer::setWilsonCoefficients`. The default WC settings are the SM. A typical example of the usage of this method is

```
    ham.setWilsonCoefficients("BtoCTauNu",
                        {{"S_qLlL", 1.}, {"T_qLlL",0.25}});
```

where the first argument can be any of `"BtoCTauNu"`, `"BtoCTMuNu"`, or `"BtoCENu"` as desired. The second argument is a `map<string, complex<double>>` of each WC to its desired value. The full list of WCs and their definitions is supplied in Sec. V B. An optional third `bool` argument may be supplied to specify whether the WCs should be set for the numerator/output (`true`) or denominator/input (`false`), with the default being `true`. As an alternative, one may instead pass as second argument a `vector<complex<double>`, corresponding to the ordered basis

```
   {"SM", "S_qLlL", "S_qRlL", "V_qLlL", "V_qRlL", "T_qLlL",
                  "S_qLlR", "S_qRlR", "V_qLlR", "V_qRlR", "T_qRlR"}.
```

As mentioned in Sec II A, certain form factor classes (e.g. those with names ending in “`Var`”) assign additional ‘error’ or ‘uncertainty’ indices to the form factors, in the sense defined by eq. (6). This generalizes the tensor weights into the form factor error eigenspace (or whatever space is defined in the relevant parametrization’s class), which may then be contracted with the desired error (eigen)vecto, permitting reweighting of the events to any point in this space. This error (eigen)vector is set via the method `Hammer::setFFEigenvectors`. The usage is similar `setWilsonCoefficients`, except that `setFFEigenvectors` takes the name of the hadronic process in `"XtoY"` form (see Sec. III D), the name of the FF parametrization, and then either a `map<string, complex<double>>` of the error coordinates to be changed, or a vector of coordinates `vector<complex<double>`, with respect to the basis defined by the parametrization’s class. A typical example of the usage of this method is

```
    ham.setFFEigenvectors("BtoD*", "BGLVar",
                        {{"delta_a1", 0.1}, {"delta_b1",-0.05}});
```

See Sec. V E for definitions of currently implemented FF error bases. Parametrizations with FF uncertainty indices are intended to be used only in the numerator FF schemes; specific settings for the denominator classes can be implemented by a duplicated FF scheme and `setOptions` (see Sec. III D).

J. Adding and retrieving histograms

Histograms of arbitrary dimensionality may be created by the `Hammer` library. In general, histogram bins contain event weight tensors, which are *direct products* of the process weight tensors for all processes in the event that are included by an `includeDecay` specification (and not specifically removed by a later `removeProcess` invocation). It is up to the user to determine programmatically which processes in an event are (or are not) included. For example, under the include specification shown in Sec. III C, an event featuring $\bar{B}^0 \to (D^{*+} \to (D^+ \to K^+\pi^+\pi^-)\gamma)(\tau^- \to \ell^-\nu\nu)$ and $B^0 \to D^-\mu^+\nu$ would have an event weight composed from the product of both process weights, while an event featuring $\bar{B}^0 \to (D^+(\tau^- \to \ell^-\nu\nu)$ and $B^0 \to D^-\mu^+\nu$ would just have an event weight equal to the process weight for the $B^0 \to D^-\mu^+\nu$ decay.

The event weight tensor may be contracted with arbitrary WCs to generate *a posteriori* the corresponding histogram bin weight. Thus once a histogram is computed, it is computed for all NP. More specifically, a histogram contains elements that are `BinContents` structs, with members `sumWi`, `sumWi2` and `n` for weight tensor, weight squared tensor and number of events in the bin.

A histogram is declared by `Hammer::addHistogram`, which takes as arguments a name string and either: a vector of dimensions, a bool for under/overflow and a vector of ranges; or a vector of bin edges and a bool for under/overflow. The method `addHistogram` does not create a single histogram, but rather a *histogram set*: A separate histogram is created for each unique event ID – a `set` of process IDs for all processes included in the event – and in turn for each FF scheme name specified by `addFFScheme`. For instance

```
ham.addHistogram("q2VsEmu", {20, 15}, false, {{3.,12.},{0,2.5}});
```

creates a *histogram set* each with 20×15 bins, no under/overflow, binned uniformly over the respective ranges 3–12 and 0–2.5 (in appropriate units). Alternatively, for non-uniform bins

```
ham.addHistogram("q2VsEmu", {{3.,5.,9.,12.},{0,1,2.5}}, true);
```

which creates a 3×2 bin tensor, with additional under/overflow bins. For an MC sample with N unique event IDs and m declared FF schemes, the above `addHistogram` invocation would create $m \times n$ unique 20×15 histograms, all with the name `"q2VsEmu"`.

Filling of histograms for a specific event is perfomed by `Hammer::fillEventHistogram`, which takes the histogram name and the values of the observables corresponding to each histogram dimension. (A deprecated method `Hammer::setEventHistogramBin` takes the indices of the bin to be filled.) For example,

```
ham.fillEventHistogram("q2VsEmu", {4., 0.5});
```

fills the appropriate bin element for the `"q2VsEmu"` histograms belonging to the event being processed, and fills the relevant histograms for each FF scheme name. Invocations of `fillEventHistogram` must occur before `Hammer::processEvent`. Otherwise, the relevant histogram will not be filled with the weight for event being processed: If `fillEventHistogram` is not invoked for a particular histogram for a particular event, the event weight is not added to the hisotgram.

In many use cases, the entire histogram set is not required, but rather its direct sum. Computing and storing only the latter permits both speed gains and space savings. The method `Hammer::collapseProcessesInHistogram` takes a name of a histogram, and causes all members of the histogram set containing the same tensor structures to be summed and collapsed into a single histogram. For instance,

```
ham.collapseProcessesInHistogram("q2VsEmu");
```

This method should invoked before `initRun`.

A single bin histogram set `"Total Sum of Weights"` may be created via the method `Hammer::addTotalSumOfWeights`, which takes additional bools for collapsing processes and uncertainties (see Sec. III K). This method should invoked before `initRun`. The `"Total Sum of Weights"` histogram, if it has been created, is automatically filled by `processEvent`.

Once all events have been processed (or if histograms are reloaded from a file) the user may retrieve a specific histogram the method `Hammer::getHistogram`, that takes a histogram name and a FF scheme name. NP choices must be specified first via `setWilsonCoefficients`, as must FF uncertainties via `setFFEigenvectors` if a parametrization in the desired FF scheme has them. For example,

```
ham.setWilsonCoefficients("BtoCTauNu",
                          {{"S_qRlL", 1.},{"S_qLlL", 0.5}});
auto histo = ham.getHistogram("q2VsEmu", "Scheme2");
```

would combine the histograms for all processed events that have been assigned a bin in the `"q2VsEmu"` histogram, then contracts the event weights with the specified NP Wilson coefficients and FF eigenvectors. By contrast, the method `getHistograms` (note the plural) extracts all histograms of a specific name and scheme. For example

```
auto histos = ham.getHistograms("q2VsEmu", "Scheme2");
```

produces a map of eventIDs to histogram for all available `"q2VsEmu"` histograms in FF scheme `"Scheme2"`.

K. Histogram specialization and uncertainties

In a specific histogram one may wish to fix *a priori* all WCs of a specific type or all FF indices for a particular scheme, in order to reduce space or reweighting times. This acheived with the methods `specializeWCInHistogram` and `specializeFFInHistogram` respectively, that take the histogram name plus the arguments required by `setWilsonCoefficients` or `setFFEigenvectors`. An example usage

```
    map<string, complex<double>> special {{"SM", 1.},
            {"S_qLlL", 0.2i}, {"T_qLlL", 0.05}};
    ham.specializeWCInHistogram("q2VsEmu", "BtoCTauNu", special);
```

would fix these specific $b \to c\tau\nu$ WCs and set all other $b \to c\tau\nu$ WCs to zero only in the "`q2VsEmu`" histogram. Other WCs, such as those for $b \to c\mu\nu$, would remain unfixed. This method should be invoked after `initRun`.

Computation of the weight-squared uncertainties (accessed from the `BinContents` struct via `sumWi2`) is off by default. This may be enabled globally via the options setting `ham.setOptions("Histos: {KeepErrors: true}")`. However, for computational speed and/or memory efficiency, it may be instead enabled or disabled for individual histograms via `Hammer::keepErrorsInHistogram`, which takes the name of the histogram as an argument, and a bool. For instance

```
    ham.keepErrorsInHistogram("q2VsEmu", true);
```

enables weight-squared computation for this particular histogram. This method should be invoked before `initRun`.

L. Pure phase space vertices

The `Hammer` library permits the user to declare particular vertices, in either the denominator or numerator amplitude, to be evaluated as pure phase space. This is achieved by the method `Hammer::addPurePSVertices`, which takes a set of string vertices as an argument, and an optional enum `WTerm` value to declare whether the evaluation should be applied to the numerator and/or denominator (numerator by default). The enum `WTerm` has values {`COMMON, NUMERATOR, DENOMINATOR`}

As an example

```
    ham.addPurePSVertices({"TauMuNuNu","D*+DPi"});
    ham.addPurePSVertices({"D*DGamma"}, WTerm::DENOMINATOR);
```

declares all $\tau \to \mu\nu\nu$ and $D^{*+} \to D\pi$ vertices in the numerator and all $D^* \to D\gamma$ vertices in the denominator, to be phase space (subject to the rules below). The equivalent initialization card definition is

```
PurePSVertices:
    Numerator: [ TauMuNuNu, D*+DPi ]
    Denominator: [ D*DGamma ]
```

The library employs the pure phase space definition

$$\frac{1}{\prod_k |\{s_k\}|} \sum_{s_i, r_j} \left|\mathcal{M}_{s_1,\ldots,s_n;r_1,\ldots,r_m}\right|^2 = 1 \times (m^{6-2n}) \,, \tag{8}$$

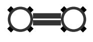
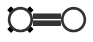
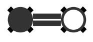
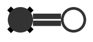
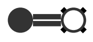
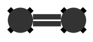
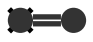
where s_i (r_i) are incoming (outgoing) quantum numbers, $|\{s_k\}|$ is the number of states of s_k, m is the mass of the parent particle in the vertex, and n the number of daughters. I.e., the squared matrix element averaged over initial states and summed over final states is set to unity times a factor that preserves dimsionality of the overall amplitude. Upon the declaration of a vertex as PS, averaging over the initial states of all immediate (non-PS) daughter vertices is automatically performed.

The declaration of a vertex as phase space within an edge may be ambiguous, if the other vertex is not declared as PS too. This ambiguity is resolved by the library by an *exclusive* implementation of the `addPurePSVertices` method, according to the following rules:

(i) If both vertices in an edge are declared as PS, the edge is set to PS.

(ii) The declaration of a single vertex in an edge as PS is obeyed only if the remaining vertex has a known vertex amplitude.

Labelling a PS declaration by an underlaid cross, i.e. or , these rules are represented as follows:

	Edge is set to PS
	Declaration refused; a warning is thrown
	Edge is set to PS
	Declaration refused; a warning is thrown
	Edge is replaced by remaining ●
	Edge is set to PS
	Edge is replaced by remaining ●

An example of these rules are shown in Table V for the examples of Table II, based on the process tree in Fig. 2. In the first example, the declaration of vertex 1 as pure phase space is accepted, with the 0–1 edge being replaced by the known vertex amplitude at vertex 0. In the second example, the declaration is refused, since vertex 5 cannot be evaluated independently.

M. PHOTOS

Typical MC samples include collinear radiative corrections, incoherently appended to the relevant vertices by the PHOTOS algorithm [3], ignoring typically negligible interference effects. Inclusion of such (typically very soft) radiative photons requires the vertex (and all daughter vertex) momenta to be rebalanced, such that overall momentum remains conserved. For the purpose of reweighting the truth level process, these photons must be pruned from the process tree, which in turn requires an reversion of the kinematic rebalancing. (As such, because they are automatically pruned, radiative photons need not be specified in `includeDecay` or `forbidDecay` specifications.)

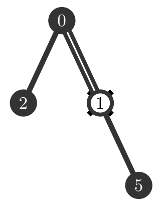
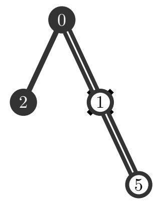
Known Amplitudes	Evaluated Amplitudes
	⓪, ②, ⑤
	⓪, ①═⑤, ②

TABLE V. Example arithmetic for filling amplitudes for the examples of Tab. II, with an additional phase space declaration on vertex 1.

The effect of the kinematic rebalancing on the actual event weight is generally negligible: The main concern is to ensure momentum conservation in the process tree once the photon is removed. With this in mind, and following the PHOTOS prescription for kinematic rebalancing [3], the `Hammer` library therefore identifies radiative photons, and reverts the kinematics to pre-radiative corrected form, by the following procedure:

(i) If a vertex contains 3 or more particles, with at least one photon, the softest photon is identified as radiative.

(ii) The radiative photon, $\gamma_{\rm rad}$, is assumed to be associated with the nearest charged particle, labelled 'ch', in the polar angle distance, $\delta\theta$.

(iii) The radiative vertex, and all daughter particles, are then partitioned into: The parent particle, 'P'; The charged particle, 'ch', and all its descendants, the 'ch subtree'; All other particles in the radiative vertex except $\gamma_{\rm rad}$, collectively called 'Y', and all their descendants, the 'Y subtree'. The radiative vertex is thus written $P \to {\rm ch} + Y + \gamma_{\rm rad}$.

(iv) The ch and Y subtrees are boosted to the $p_{\rm ch}+p_Y$ rest frame, $R_{{\rm ch}+Y}$, so that necessarily $\boldsymbol{p}_{\rm ch}$ and $\boldsymbol{p}_Y$ are back-to-back.

(v) In $R_{{\rm ch}+Y}$ frame, writing $p_Y = (E_Y, \boldsymbol{p}_Y)$ and $p_{\rm ch} = (E_{\rm ch}, \boldsymbol{p}_{\rm ch})$, the ch subtree and Y subtree are then *independently* longitudinally boosted by

$$\beta\gamma_{\rm ch} = \frac{E_{\rm ch}|\boldsymbol{p}^*| - E^*_{\rm ch}|\boldsymbol{p}_{\rm ch}|}{m^2_{\rm ch}}\,, \qquad \beta\gamma_Y = \frac{E_Y|\boldsymbol{p}^*| - E^*_Y|\boldsymbol{p}_Y|}{m^2_Y}\,, \tag{9}$$

in which the starred quantities are the usual P rest frame kinematic objects for the two body decay $P \to \text{ch} + Y$, i.e.

$$E^*_{\text{ch}} = \frac{m_P^2 - m_Y^2 + m_{\text{ch}}^2}{2m_P}\,, \qquad E^*_Y = \frac{m_P^2 - m_{\text{ch}}^2 + m_Y^2}{2m_P}\,, \qquad |\boldsymbol{p}^*| = \frac{m_P}{2}\lambda^{1/2}\left[\frac{m_Y}{m_P}, \frac{m_{\text{ch}}}{m_P}\right], \tag{10}$$

with $\lambda(x, y) = (1-(x+y)^2)(1-(x-y)^2)$. Under these independent boosts, momentum conservation is restored to the $P \to \text{ch} + Y$ vertex with γ_{rad} removed.

(vi) The ch and Y subtrees are then boosted to the frame such that $p_{\text{ch}} + p_Y = p_P$, the latter meaning the actual momentum of particle P in the process tree.

(vii) This process is repeated until (i) is no longer true.

N. Rates

The library provides the means to compute the partial width for a particular vertex via `Hammer::getRate`, which takes as argument either a vertex string or the parent and daughter PDG codes, plus a scheme. It may also take a vertex hashID, obtainable from a specific process via `Process::getVertexId`. Partial widths are returned in the units specified by `Hammer::setUnits`; the default is GeV (see Sec. III F). For example

```
ham.getRate(511, {-413, -14, 13}, "Scheme2");
ham.getRate("B0D*-MuNu", "Scheme2");
```

both return the partial width for the $B^0 \to D^{*-}\mu^+\nu$ vertex, using the form factor parameterization specified in `"Scheme2"`, and using whatever WCs or FF uncertainties have been specified. At present, the `getRate` method is charge conjugate sensitive, so in a vertex string one must specify sufficient charges to make the vertex charge unique. (For example, writing just `"B0D*MuNu"` would have corresponded to not only $B^0 \to D^{*-}\mu^+\nu$, but also the (very heavily suppressed) process $B^0 \to D^+\mu^-\bar{\nu}$.) The method `getDenominatorRate` takes just the vertex argument, and returns the partial width according to the specified denominator/input FF parametrization chosen in `setFFInputScheme`, and the denominator/input WCs.

Vertices involving new physics and/or form factor parametrizations have rates implemented in dedicated classes, and integrated over q^2 (and other invariants as needed) via Gaussian quadrature. Other partial widths, e.g. for $D^* \to D\pi$ or $\tau \to \ell\nu\nu$, are obtained from the SM branching ratios and widths specified in the `Pdg` class. The partial width for each unique vertex is computed only *once* per run, being computed and stored the first time each unique vertex in encountered in a process. Rates are computed vertex-wise inside edges. Hence e.g. while an edge ●═○ is computed as a single amplitude, the rates for the known and unknown vertices are computed and stored independently. If a vertex is set to pure PS (or successfully set to pure PS inside an edge, see Sec. III L) then following the PS definition (8) the returned rate for that vertex is the phase space rate

$$\Gamma_n^{\mathcal{PS}} = \frac{1}{2m} \int m^{6-2n} d\mathcal{PS}_n \,. \tag{11}$$

IV. THE `HAMMER` BUFFER

`Hammer` provides the ability to store header settings, generated event weights, histograms, and/or rates in binary buffers for later retrieval and reprocessing. These buffers are built on the cross-platform serialization library `flatbuffers`: The buffer structs `Hammer::IOBuffer` and `Hammer::RootIOBuffer` permit writing/reading of binary files of `Hammer` internal and `root` objects, respectively.

A. Saving

In order to save a buffer, an `ofstream` outfile must first be designated. For example,

```
ofstream outFile("./DemoSave.dat",ios::binary);
```

The methods `Hammer::save...` return a `IOBuffer`, which can be stored as sequential records in the buffer via an ostream operator. For example,

```
outFile << ham.saveRunHeader();
```

writes the declared run header, with all its settings, into an `IOBuffer` and passes it as a record into the buffer. The available record types are labelled by an enum `char Hammer::RecordType` with values `UNDEFINED = 'u'`, `HEADER = 'b'`, `EVENT = 'e'`, `HISTOGRAM = 'h'`, `RATE = 'r'`. (Note the `saveOptionCard` method instead takes a filename and a bool for whether to write default values (`true`, default) versus modified settings (`false`). Output is written in text to the specified file.)

The method `saveEventWeights` saves the event weights of the currently initialized and processed event (there may be multiple weight saved if there are multiple processes in the event). This method should be invoked only after `processEvent`. Similarly, `saveRates` writes *all* rates computed during the event loop. By contrast, the method `saveHistogram` takes a histogram name as an argument, and saves only the specified histogram set. For example,

```
outFile << ham.saveHistogram("q2VsEmu");
```

saves all the unique `"q2VsEmu"` histograms, corresponding to the unique event IDs and declared FF schemes, subject to compression settings (see Sec. III J). Any combination of these save methods may be invoked, in any order.

Saving a buffer in `root` format is achieved by passing the `IOBuffer` output of the `save...` methods into a `RootIOBuffer`, that may then be stored in a `root TTree`. Explicit implementations of this functionality are provided in various `demo...root.cc` example programs.

B. Reloading

Reloading requires creation of an `IOBuffer` into which buffer records may be loaded from a declared `ifstream` infile, via an `istream` operator. For example,

```
ifstream inFile("./DemoSave.dat", ios::binary);
Hammer::IOBuffer buf{Hammer::RecordType::UNDEFINED, 0ul, nullptr};
inFile >> buf;
ham.loadRunHeader(buf);
```

attempts to load the first buffer record as a run header (returning `false` if this record is of a different type). Similarly,

```
inFile >> buf;
while(buf.kind == Hammer::RecordType::HISTOGRAM) {
    ham.loadHistogram(buf);
    inFile >> buf;
}
```

reads through the entire buffer, with the method `loadHistogram` loading all the histogram records that are found. One may then subsequently invoke `getHistogram` for a desired histogram as described in Sec. III J. The method `loadRates` behaves similarly. Event weights can be reloaded via `loadEventWeights`, recreating the original event loop provided `initEvent` and `processEvent` are called appropriately. For example,

```
inFile >> buf;
while(buf.kind == Hammer::RecordType::EVENT) {
    ham.initEvent();
    ham.loadEventWeights(buf);
    double q2 = ...;
    ham.fillEventHistogram("Q2", {q2});
    ham.processEvent();
    inFile >> buf;
}
```

would permit reprocessing of saved event weights into a newly created `"Q2"` histogram.

Loading a buffer in `root` format is achieved by reading the `RootIOBuffer` stored in a `TTree` into an `IOBuffer` that can be passed to the `load...` methods. Explicit implementations of this functionality are provided in various `demo...root.cc` example programs.

C. Parallelization and merging

In order to permit parallelization of initialization analyses, the `load...` methods accept an additional bool, to specify whether to merge the buffer contents with existing objects (`true`), or overwrite them (`false`, default).

Merging of histograms occurs if two histograms are loaded with a matching name. This merging is additive for unique histograms in each histogram set – i.e. with the same event ID and FF scheme – and otherwise results in the new unique histograms being appended to the existing histogram set. Errors are thrown if the matching histograms do not have compatible shapes or bin contents. For instance, if the `"q2VsEmu"` histogram is loaded via

```
inFile1 >> buf;
ham.loadHistogram(buf);
```

subsequently loading an identically named histogram from a second infile via

```
inFile2 >> buf;
ham.loadHistogram(buf, true);
```

will merge the two histograms together according to the above rules. It is important that the merge bool is set to `false` (the default) for loading from the first file.

The methods `loadEventWeights` and `loadRates` behave similarly. For weights (rates) with matching process ID (event ID), merging permits appending of process weights (rates) computed with new form factor schemes to the process weights (rates). Errors are thrown if an FF scheme by the same name already exists. Finally, `loadRunHeader` permits merging of two sets of header settings into their union, with errors thrown for matching settings with non-matching values.

V. CONVENTIONS

A. V_{cb}

The V_{cb} prefactor is generally not included explicitly in the $b \to c$ amplitudes, form factor parameters or rates. (One exception is the BGL parametrization, whose parameters typically absorb a factor of $V_{cb}\eta_{\mathrm{EW}}$. In order to preserve uniformity among the form factor schemes, this factor is divided out of the BGL form factors.)

B. NP operator basis

A complete basis for the four-Fermi operators mediating $b \to c\bar{\ell}\nu$ decay, including right-handed neutrinos, is shown in Table VI. The NP couplings to the quark and lepton currents are denoted by χ^i_j and λ^i_j, respectively, and may in general be complex numbers. The lower index of λ denotes the ν helicity and the lower index of χ is that of the b quark. The NP couplings are normalized with respect to the SM current.

Current	`WC Tag`	WC	4-Fermi/$(i2\sqrt{2}\,V_{cb}G_F)$
SM	`SM`	1	$\big[\bar{c}\gamma^\mu P_L b\big]\big[\bar{\ell}\gamma_\mu P_L\nu\big]$
Vector	`V_qLlL`	$\chi^V_L\lambda^V_L$	$\big[\bar{c}\chi^V_L\gamma^\mu P_L b\big]\big[\bar{\ell}\lambda^V_L\gamma_\mu P_L\nu\big]$
	`V_qRlL`	$\chi^V_R\lambda^V_L$	$\big[\bar{c}\chi^V_R\gamma^\mu P_R b\big]\big[\bar{\ell}\lambda^V_L\gamma_\mu P_L\nu\big]$
	`V_qLlR`	$\chi^V_L\lambda^V_R$	$\big[\bar{c}\chi^V_L\gamma^\mu P_L b\big]\big[\bar{\ell}\lambda^V_R\gamma_\mu P_R\nu\big]$
	`V_qRlR`	$\chi^V_R\lambda^V_R$	$\big[\bar{c}\chi^V_R\gamma^\mu P_R b\big]\big[\bar{\ell}\lambda^V_R\gamma_\mu P_R\nu\big]$
Scalar	`S_qLlL`	$\chi^S_L\lambda^S_L$	$\big[\bar{c}\chi^S_L P_L b\big]\big[\bar{\ell}\lambda^S_L P_L\nu\big]$
	`S_qRlL`	$\chi^S_R\lambda^S_L$	$\big[\bar{c}\chi^S_R P_R b\big]\big[\bar{\ell}\lambda^S_L P_L\nu\big]$
	`S_qLlR`	$\chi^S_L\lambda^S_R$	$\big[\bar{c}\chi^S_L P_L b\big]\big[\bar{\ell}\lambda^S_R P_R\nu\big]$
	`S_qRlR`	$\chi^S_R\lambda^S_R$	$\big[\bar{c}\chi^S_R P_R b\big]\big[\bar{\ell}\lambda^S_R P_R\nu\big]$
Tensor	`T_qLlL`	$\chi^T_L\lambda^T_L$	$\big[\bar{c}\,\chi^T_L\sigma^{\mu\nu} P_L b\big]\big[\bar{\ell}\,\lambda^T_L\sigma_{\mu\nu} P_L\nu\big]$
	`T_qRlR`	$\chi^T_R\lambda^T_R$	$\big[\bar{c}\,\chi^T_R\sigma^{\mu\nu} P_R b\big]\big[\bar{\ell}\,\lambda^T_R\sigma_{\mu\nu} P_R\nu\big]$

TABLE VI. NP operator basis, and coupling conventions.

These conventions correspond to the conventions of Refs. [1] via

$$\begin{aligned} \chi^V_L &= \alpha^{V*}_L\,, & \chi^V_R &= \alpha^{V*}_R\,,\\ \chi^S_R &= -\alpha^{S*}_L\,, & \chi^S_L &= -\alpha^{S*}_R\,,\\ \chi^T_R &= -\alpha^{T*}_L\,, & \chi^T_L &= -\alpha^{T*}_R\,,\\ \lambda^{V,S,T}_L &= \beta^{V,S,T*}_L\,, & \lambda^{V,S,T}_R &= \beta^{V,S,T*}_R\,. \end{aligned} \tag{12}$$

All internal `Hammer` calculations are done in the $\alpha^i_j\beta^k_l$ basis of Ref. [1], which is naturally defined for $\bar{b} \to \bar{c}\ell\nu$ transitions and their corresponding $\bar{b}\Gamma c$ operators. Since, however, specification of WCs with respect to $\bar{c}\Gamma b$ operators is the predominant convention, `Hammer` inputs are specified in the $\chi^i_j\lambda^k_l$ WC basis. In the conventions of Ref. [2], $\chi = \tilde{\alpha}$, and $\lambda = \tilde{\beta}$, but we discard this tilded notation hereafter, so that there is no potential confusion as to which convention the WC tag subscripts, '`_qXlX`', adhere.

C. Lorentz signs

For all amplitudes encoded into `Hammer`, we use a trace -2 metric, and the Lorentz sign conventions

$$\mathrm{Tr}[\gamma^\mu\gamma^\nu\gamma^\sigma\gamma^\rho\gamma^5] = -4i\epsilon^{\mu\nu\rho\sigma}\,, \qquad \epsilon^{0123} = +1\,. \tag{13}$$

These choices fully specify all other possible ambiguous signs, for example the γ^5 trace choice is equivalent to $\sigma^{\mu\nu}\gamma^5 \equiv +\frac{i}{2}\,\epsilon^{\mu\nu\rho\sigma}\sigma_{\rho\sigma}$, with $\sigma_{\mu\nu} = \frac{i}{2}[\gamma^\mu, \gamma^\nu]\,.$

D. Form Factors and Maps

1. $\overline{B} \to D$

The $\overline{B} \to D$ form factor tensor has ordered components

$$\mathrm{FF}_D = \Big\{f_S,\, f_0,\, f_+,\, f_T\Big\}\,, \tag{14}$$

which are defined via

$$\big\langle D\big|\,\bar{c}\,b\,\big|\overline{B}\big\rangle \equiv f_S\,, \tag{15a}$$

$$\big\langle D\big|\,\bar{c}\gamma^\mu b\,\big|\overline{B}\big\rangle \equiv f_+(p_B+p_D)^\mu + [f_0 - f_+]\,\frac{m_B^2 - m_D^2}{q^2}\,q^\mu\,, \tag{15b}$$

$$\big\langle D\big|\,\bar{c}\sigma^{\mu\nu} b\,\big|\overline{B}\big\rangle \equiv i f_T\Big[(p_B+p_D)^\mu q^\nu - (p_B+p_D)^\nu q^\mu\Big]\,. \tag{15c}$$

These definitions map to the conventional dimensionless form factor set h_S, h_+, h_-, h_T, as defined in e.g. Ref. [4], via

$$f_S = \sqrt{r_D}(w+1)m_B h_S\,, \tag{16a}$$

$$f_0 = \frac{\sqrt{r_D}}{r_D^2 - 1}\big[(r_D+1)(w-1)h_- + (r_D-1)(w+1)h_+\big] \tag{16b}$$

$$f_+ = \frac{(r_D-1)h_- + (r_D+1)h_+}{2\sqrt{r_D}}\,, \tag{16c}$$

$$f_T = \frac{h_{\mathrm{T}}}{2\sqrt{r_D}m_B}\,, \tag{16d}$$

with $r_D = m_D/m_B$. The $\overline{B} \to D$ form factors h_i are defined under the sign convention $\mathrm{Tr}[\gamma^\mu\gamma^\nu\gamma^\sigma\gamma^\rho\gamma^5] = +4i\epsilon^{\mu\nu\rho\sigma}$, which is accounted for in eqs. (16).

2. $\overline{B} \to D^*$

The $\overline{B} \to D^*$ form factor tensor has ordered components

$$\mathrm{FF}_{D^*} = \Big\{a_0,\, f,\, g,\, a_-,\, a_+,\, a_{T_0},\, a_{T_-},\, a_{T_+}\Big\}\,, \tag{17}$$

which are defined via

$$\big\langle D^*\big|\,\bar{c}\gamma^5 b\,\big|\overline{B}\big\rangle \equiv a_0\,\varepsilon^*\!\cdot p_B\ , \tag{18a}$$

$$\big\langle D^*\big|\,\bar{c}\gamma^\mu b\,\big|\overline{B}\big\rangle \equiv -ig\,\epsilon^{\mu\nu\rho\sigma}\,\varepsilon^*_\nu\,(p_B+p_{D^*})_\rho\,q_\sigma\,, \tag{18b}$$

$$\big\langle D^*\big|\,\bar{c}\gamma^\mu\gamma^5 b\,\big|\overline{B}\big\rangle \equiv \varepsilon^{*\mu} f + a_+\,\varepsilon^*\!\cdot p_B\,(p_B+p_{D^*})^\mu + a_-\,\varepsilon^*\!\cdot p_B\,q^\mu\,, \tag{18c}$$

$$\begin{aligned}\big\langle D^*\big|\,\bar{c}\sigma^{\mu\nu} b\,\big|\overline{B}\big\rangle \equiv{}& -a_{T_+}\,\epsilon^{\mu\nu\rho\sigma}\varepsilon^*_\rho(p_B+p_{D^*})_\sigma - a_{T_-}\,\epsilon^{\mu\nu\rho\sigma}\varepsilon^*_\rho\,q_\sigma\\ & - a_{T_0}\,\varepsilon^*\!\cdot p_B\,\epsilon^{\mu\nu\rho\sigma}(p_B+p_{D^*})_\rho\,q_\sigma\,.\end{aligned} \tag{18d}$$

These definitions map to the conventional dimensionless form factor set $h_P, h_V, h_{A_{1,2,3}}, h_{T_{1,2,3}}$, as defined in e.g. Ref. [4], via

$$a_0 = -\sqrt{r_{D^*}}h_P\,, \tag{19a}$$

$$f = \sqrt{r_{D^*}}(w+1)m_B h_{A_1}\,, \tag{19b}$$

$$g = \frac{h_{\rm V}}{2\sqrt{r_{D^*}}m_B}\,, \tag{19c}$$

$$a_- = \frac{h_{A_3} - r_{D^*}h_{A_2}}{2\sqrt{r_{D^*}}m_B}\,, \tag{19d}$$

$$a_+ = -\frac{r_{D^*}h_{A_2} + h_{A_3}}{2\sqrt{r_{D^*}}m_B}\,, \tag{19e}$$

$$a_{T_0} = \frac{h_{T_3}}{2\sqrt{r_{D^*}}m_B^2}\,, \tag{19f}$$

$$a_{T_-} = \frac{(1-r_{D^*})h_{T_1} - (r_{D^*}+1)h_{T_2}}{2\sqrt{r_{D^*}}}\,, \tag{19g}$$

$$a_{T_+} = \frac{(1-r_{D^*})h_{T_2} - (r_{D^*}+1)h_{T_1}}{2\sqrt{r_{D^*}}}\,. \tag{19h}$$

with $r_{D^*} = m_{D^*}/m_B$. The $\overline{B}\to D^*$ form factors h_i are defined under the sign convention $\mathrm{Tr}[\gamma^\mu\gamma^\nu\gamma^\sigma\gamma^\rho\gamma^5] = +4i\epsilon^{\mu\nu\rho\sigma}$, which is accounted for in eqs. (19).

3. $B\to D^{**}$

The $B\to D^{**}$ form factor tensors are ordered

$$\mathrm{FF}_{D_0^*} = \Big\{g_P,\, g_+,\, g_-,\, g_T\Big\}\,, \tag{20a}$$

$$\mathrm{FF}_{D_1^*} = \Big\{g_S,\, g_{V_1},\, g_{V_2},\, g_{V_3},\, g_a,\, g_{T_1},\, g_{T_2},\, g_{T_3}\Big\}\,, \tag{20b}$$

$$\mathrm{FF}_{D_1} = \Big\{f_S,\, f_{V_1},\, f_{V_2},\, f_{V_3},\, f_a,\, f_{T_1},\, f_{T_2},\, f_{T_3}\Big\}\,, \tag{20c}$$

$$\mathrm{FF}_{D_2^*} = \Big\{k_P,\, k_{A_1},\, k_{A_2},\, k_{A_3},\, k_V,\, k_{T_1},\, k_{T_2},\, k_{T_3}\Big\}\,, \tag{20d}$$

which following Ref. [2], are defined for $\overline{B}\to D_0^*$ via

$$\big\langle D_0^*\big|\,\bar{c}\,b\,\big|B\big\rangle = \big\langle D_0^*\big|\,\bar{c}\gamma_\mu b\,\big|B\big\rangle = 0\,,$$

$$
\begin{aligned}
\big\langle D_0^*\big|\, \bar c\gamma_5 b \,\big|B\big\rangle &= \sqrt{m_{D_0^*} m_B}\, g_P \,,\\
\big\langle D_0^*\big|\, \bar c\gamma_\mu\gamma_5 b \,\big|B\big\rangle &= \sqrt{m_{D_0^*} m_B}\big[g_+(v_\mu + v'_\mu) + g_-(v_\mu - v'_\mu)\big],\\
\big\langle D_0^*\big|\, \bar c\sigma_{\mu\nu} b \,\big|B\big\rangle &= \sqrt{m_{D_0^*} m_B}\, g_T\, \varepsilon_{\mu\nu\alpha\beta}\, v^\alpha v'^\beta \,,
\end{aligned} \tag{21a}
$$

for $\overline{B} \to D_1^*$,

$$
\begin{aligned}
\big\langle D_1^*\big|\, \bar c\, b \,\big|B\big\rangle &= -\sqrt{m_{D_1^*} m_B}\, g_S\, (\epsilon^* \cdot v) \,,\\
\big\langle D_1^*\big|\, \bar c\gamma_5 b \,\big|B\big\rangle &= 0 \,,\\
\big\langle D_1^*\big|\, \bar c\gamma_\mu b \,\big|B\big\rangle &= \sqrt{m_{D_1^*} m_B}\, \big[g_{V_1}\, \epsilon^*_\mu + (g_{V_2} v_\mu + g_{V_3} v'_\mu)\, (\epsilon^* \cdot v)\big] \,,\\
\big\langle D_1^*\big|\, \bar c\gamma_\mu\gamma_5 b \,\big|B\big\rangle &= i\sqrt{m_{D_1^*} m_B}\, g_A\, \varepsilon_{\mu\alpha\beta\gamma}\, \epsilon^{*\alpha} v^\beta\, v'^\gamma \,,\\
\big\langle D_1^*\big|\, \bar c\sigma_{\mu\nu} b \,\big|B\big\rangle &= i\sqrt{m_{D_1^*} m_B}\, \big[g_{T_1}(\epsilon^*_\mu v_\nu - \epsilon^*_\nu v_\mu) + g_{T_2}(\epsilon^*_\mu v'_\nu - \epsilon^*_\nu v'_\mu) + g_{T_3}(\epsilon^* {\cdot} v)(v_\mu v'_\nu - v_\nu v'_\mu)\big].
\end{aligned} \tag{21b}
$$

for $\overline{B} \to D_1$,

$$
\begin{aligned}
\big\langle D_1\big|\, \bar c\, b \,\big|B\big\rangle &= \sqrt{m_{D_1} m_B}\, f_S\, (\epsilon^* \cdot v) \,,\\
\big\langle D_1\big|\, \bar c\gamma_5 b \,\big|B\big\rangle &= 0 \,,\\
\big\langle D_1\big|\, \bar c\gamma_\mu b \,\big|B\big\rangle &= \sqrt{m_{D_1} m_B}\, \big[f_{V_1}\, \epsilon^*_\mu + (f_{V_2} v_\mu + f_{V_3} v'_\mu)(\epsilon^* \cdot v)\big],\\
\big\langle D_1\big|\, \bar c\gamma_\mu\gamma_5 b \,\big|B\big\rangle &= i\, \sqrt{m_{D_1} m_B}\, f_A\, \varepsilon_{\mu\alpha\beta\gamma} \epsilon^{*\alpha} v^\beta v'^\gamma \,,\\
\big\langle D_1\big|\, \bar c\sigma_{\mu\nu} b \,\big|B\big\rangle &= i\sqrt{m_{D_1} m_B}\big[f_{T_1}(\epsilon^*_\mu v_\nu - \epsilon^*_\nu v_\mu) + f_{T_2}(\epsilon^*_\mu v'_\nu - \epsilon^*_\nu v'_\mu) + f_{T_3}(\epsilon^* {\cdot} v)(v_\mu v'_\nu - v_\nu v'_\mu)\big],
\end{aligned} \tag{21c}
$$

and finally for $\overline{B} \to D_2^*$,

$$
\begin{aligned}
\big\langle D_2^*\big|\, \bar c\, b \,\big|B\big\rangle &= 0 \,,\\
\big\langle D_2^*\big|\, \bar c\gamma_5 b \,\big|B\big\rangle &= \sqrt{m_{D_2^*} m_B}\, k_P\, \epsilon^*_{\alpha\beta}\, v^\alpha v^\beta \,,\\
\big\langle D_2^*\big|\, \bar c\gamma_\mu b \,\big|B\big\rangle &= i\sqrt{m_{D_2^*} m_B}\, k_V\, \varepsilon_{\mu\alpha\beta\gamma}\, \epsilon^{*\alpha\sigma} v_\sigma v^\beta v'^\gamma \,,\\
\big\langle D_2^*\big|\, \bar c\gamma_\mu\gamma_5 b \,\big|B\big\rangle &= \sqrt{m_{D_2^*} m_B}\, \big[k_{A_1}\, \epsilon^*_{\mu\alpha} v^\alpha + (k_{A_2} v_\mu + k_{A_3} v'_\mu)\, \epsilon^*_{\alpha\beta}\, v^\alpha v^\beta\big],\\
\big\langle D_2^*\big|\, \bar c\sigma_{\mu\nu} b \,\big|B\big\rangle &= \sqrt{m_{D_2^*} m_B}\, \varepsilon_{\mu\nu\alpha\beta}\, \big\{\big[k_{T_1}(v + v')^\alpha + k_{T_2}(v - v')^\alpha)\big]\, \epsilon^{*\gamma\beta} v_\gamma + k_{T_3}\, v^\alpha v'^\beta \epsilon^{*\rho\sigma} v_\rho v_\sigma\big\}.
\end{aligned} \tag{21d}
$$

4. $\Lambda_b \to \Lambda_c$

The $\Lambda_b^0 \to \Lambda_c^+$ form factor tensor has ordered components

$$
\mathrm{FF}_{\Lambda_c} = \Big\{h_S,\, h_P,\, f_1,\, f_2,\, f_3,\, g_1,\, g_2,\, g_3,\, h_1,\, h_2,\, h_3,\, h_4\Big\}\,, \tag{22}
$$

which are defined as in Ref. Ref. [5], [], under the sign convention $\mathrm{Tr}[\gamma^\mu\gamma^\nu\gamma^\sigma\gamma^\rho\gamma^5] = -4i\epsilon^{\mu\nu\rho\sigma}$, via

$$
\langle\Lambda_c(p', s')|\bar c\, b|\Lambda_b(p, s)\rangle = h_S\, \bar u(p', s')\, u(p, s)\,, \tag{23a}
$$

$$
\langle\Lambda_c(p', s')|\bar c\gamma_5 b|\Lambda_b(p, s)\rangle = h_P\, \bar u(p', s')\, \gamma_5\, u(p, s)\,, \tag{23b}
$$

$$
\langle\Lambda_c(p', s')|\bar c\gamma_\nu b|\Lambda_b(p, s)\rangle = \bar u(p', s')\big[f_1\gamma_\mu + f_2 v_\mu + f_3 v'_\mu\big] u(p, s)\,, \tag{23c}
$$

$$\langle \Lambda_c(p',s')|\bar{c}\gamma_\nu\gamma_5 b|\Lambda_b(p,s)\rangle = \bar{u}(p',s')\big[g_1\gamma_\mu + g_2 v_\mu + g_3 v'_\mu\big]\gamma_5\, u(p,s)\,, \tag{23d}$$

$$\langle \Lambda_c(p',s')|\bar{c}\,\sigma_{\mu\nu}\,b|\Lambda_b(p,s)\rangle = \bar{u}(p',s')\big[h_1\,\sigma_{\mu\nu} + i\,h_2(v_\mu\gamma_\nu - v_\nu\gamma_\mu) + i\,h_3(v'_\mu\gamma_\nu - v'_\nu\gamma_\mu) \tag{23e}$$

$$+\, i\,h_4(v_\mu v'_\nu - v_\nu v'_\mu)\big]u(p,s)\,. \tag{23f}$$

The spinors are normalized to $\bar{u}(p,s)u(p,s) = 2m$. (Note that another common definition for the SM form factors is [6]

$$\langle \Lambda_c(p',s')|\bar{c}\gamma_\mu b|\Lambda_b(p,s)\rangle = \bar{u}(p',s')\big[F_1\,\gamma_\mu - iF_2\,\sigma_{\mu\nu}\,q^\nu + F_3\,q_\mu\big]u(p,s)\,,$$
$$\langle \Lambda_c(p',s')|\bar{c}\gamma_\mu\gamma_5 b|\Lambda_b(p,s)\rangle = \bar{u}(p',s')\big[G_1\,\gamma_\mu - iG_2\,\sigma_{\mu\nu}\,q^\nu + G_3\,q_\mu\big]\gamma_5\,u(p,s)\,. \tag{24}$$

The notation of Ref. [6] also exchanges upper and lowercase symbols – i.e. $F_i \leftrightarrow f_i$ and $G_i \leftrightarrow g_i$ – with respect to Eqs. (23) and (24).)

E. Form Factor Uncertainty Classes

At present, the `BGLVar` parametrization permits via `setFFEignevectors` functionality (see Sec. III I), direct manipulation of the a_i, b_i, c_i and d_i parameters for $B \to D^*$ (also denoted a_i^g, a_i^f, $a_i^{\mathcal{F}_1}$ and $a_i^{\mathcal{P}_1}$, respectively, in some notational conventions), with respect to the basis

```
{"delta_a0","delta_a1","delta_a2","delta_b0","delta_b1","delta_b2",
            "delta_c1","delta_c2", "delta_d0","delta_d1"} ,
```

as well as the $a_i^{f_+}$ and $a_i^{f_0}$, $i = 0, \ldots, 3$ parameters for $B \to D$ (also denoted a_i and b_i, respectively, in some notational conventions), with respect to the basis

```
{"delta_ap0","delta_ap1","delta_ap2","delta_ap3",
            "delta_a00","delta_a01","delta_a02","delta_a03"} .
```

F. Kinematics and phase space

G. τ spinors

H. $D^{(*,**)}$ polarizations, Λ_c spins

VI. INSTALLATION

The `Hammer` package can be installed from the source code. The most recent version is available at:

Before compiling the code, the folliwing dependency requirements should be met:

- `boost` ver. ≥ 1.50

- `cmake` ver. ≥ 3.2
- `yaml-cpp` ver. ≥ 0.5
- a C++ compiler supporting C++14 (e.g. `gcc` ver. ≥ 5.1 or `clang` ver. ≥ 3.4)
- (optional) `python2` ver. ≥ 2.7 and the `Cython` python package to create the `Hammer` python package
- (optional) `ROOT` to enable `Hammer` ROOT histograms support
- (optional) `HepMC` ver. ≥ 2.06 to compile and run the examples
- (optional) `doxygen` to produce the code documentation (together with `graphviz` and optionally `LaTeX` and `doxypypy` Python package)

such packages are usually readily installed with the standard package managers provided by the operating system. For example, on Fedora Core using `dnf` one would need to install: `boost`, `cmake`, `yaml-cpp`, `yaml-cpp-devel`, (and optionally) `python-devel`, `python2-Cython`, `doxygen`, `root`, `HepMC`, `HepMC-devel`. On Ubuntu using `apt` the package needed would be: `libboost-dev`, `libyaml-cpp-dev`, `root-system`, `libhepmc-dev`, `python-pip`, and then `Cython` and `doxypypy` (with `pip install <package>`). Similarly under `MacOS` using the `homebrew` package manager one would need to install `boost`, `cmake`, `yaml-cpp`, `root6`, `cython`, `doxygen`, `hepmc` (which is provided by the `homebrew-hep` tap). Alternatively, some smaller dependencies can be installed automatically with `Hammer` during the installation process (see below for the list of dependency and the configure syntax).

Once the dependencies are installed one can expand the `Hammer` sources tarball in a temporary directory (which we will indicate as `<source_dir>` below), create a temporary build directory (`<build_dir>`) and then issue

```
> cd <build_dir>
> cmake -DCMAKE_INSTALL_PREFIX=<install_dir> <other_options> <source_dir>
> make all
> make install
```

if the directory prefix for the installation path is omitted CMake will automatically use `/usr/local`. If the unit tests are enabled (see below) one can run them in the build directory by running

```
> ctest -V
```

this is useful for checking that `Hammer` has been built properly. The main `<other_options>` are:

- `-DWITH_ROOT=[ON,OFF]`: enables the Hammer interface with ROOT,

- `-DWITH_PYTHON=[ON,OFF]`: enables the Hammer python bindings,
- `-DWITH_EXAMPLES=[ON,OFF]`: compiles and install Hammer examples and demo programs (requires HepMC),
- `-DBUILD_DOCUMENTATION=[ON,OFF]`: builds Hammer documentation pages using Doxygen,
- `-DENABLE_TESTS=[ON,OFF]`: compiles a suite of unit tests for the Hammer library.

where the default values have been underlined. If the examples are enabled, during the configuration steps a few event files necessary to run the examples programs and too large to be distributed with the source code will be automatically downloaded. Finally, after installation the examples will be located in `<install_dir>/share/Hammer/examples`. In order to facilitate installation on systems where the dependencies are either missing or not automatically recognized, the following options are available:

- `-DINSTALL_EXTERNAL_DEPENDENCIES=[ON,OFF]`: installs the missing dependencies (namely `boost`, `yaml-cpp`, `HepMC`, `Cython` and/or `doxypypy`)
- `-DFORCE_BOOST_INSTALL=[ON,OFF]`: forces Hammer to use a local `boost` installation, irrespective of whether it is already present on the system
- `-DFORCE_YAMLCPP_INSTALL=[ON,OFF]`: same as above but for `yaml-cpp`
- `-DFORCE_HEPMC_INSTALL=[ON,OFF]`: same as above but for `HepMC`

[1] Z. Ligeti, M. Papucci, and D. J. Robinson, JHEP **01**, 083 (2017), arXiv:1610.02045 [hep-ph].

[2] F. U. Bernlochner, Z. Ligeti, and D. J. Robinson, (2017), arXiv:1711.03110 [hep-ph].

[3] E. Barberio, B. van Eijk, and Z. Was, Computer Physics Communications **66**, 115 (1991).

[4] F. U. Bernlochner, Z. Ligeti, M. Papucci, and D. J. Robinson, Phys. Rev. **D95**, 115008 (2017), arXiv:1703.05330 [hep-ph].

[5] A. V. Manohar and M. B. Wise, Camb. Monogr. Part. Phys. Nucl. Phys. Cosmol. **10**, 1 (2000).

[6] A. F. Falk and M. Neubert, Phys. Rev. **D47**, 2982 (1993), arXiv:hep-ph/9209269 [hep-ph].