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

ver. 1.3.0

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

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

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

[3] *Burke Institute for Theoretical Physics, California Institute of Technology, Pasadena, CA 91125, USA*

Abstract

The manual...

CONTENTS

I. INTRODUCTION

Precision analyses of semileptonic b-hadron decays typically rely on detailed numerical Monte Carlo (MC) simulations of detector responses and acceptances. Combined with the underlying theoretical models, these simulations provide MC *templates* that may be used in fits, to translate experimental yields into theoretically well-defined parameters. This translation though can become sensitive to the template and its underlying theoretical model, introducing biases whenever there is a mismatch between the theoretical assumptions used to measure a parameter and subsequent theoretical interpretations of the data.

Such biases are known to arise in the analyses of semileptonic decays of b hadrons, in particular, for the measurements of the CKM element $|V_{cb}|$, and the ratio of semitauonic vs. semileptonic decays to light leptons,

$$R(H_c) = \frac{\Gamma(H_b \to H_c \tau \bar{\nu})}{\Gamma(H_b \to H_c l \bar{\nu})}\,, \qquad l = \mu,\, e\,, \tag{1}$$

where $H_{b,c}$ denote b- and c-flavor hadrons. To avoid this, the size of these biases need to be either carefully controlled when experiments quote their results by reversing detector effects, or they can be avoided by using dedicated MC samples for each theoretical model the measurement is confronted with. This manual presents a detailed overview of the capabilities and application programming interface of `Hammer` (*Helicity Amplitude Module for Matrix Element Reweighting*), designed expressly for the latter purpose.

Semitauonic b hadron decays have long been known to be sensitive to new physics [1–7], and were first constrained at LEP [8]. At present, the measurements of the $R(D^{(*)})$ ratios show about a 3σ tension with SM predictions, when the D and D^* modes are combined [9]. In the future, much more precise measurements of semitauonic decays are expected, not only for the $B \to D^{(*)}\tau\bar{\nu}$ channels, but also for the not yet studied decay modes, $\Lambda_b \to \Lambda_c\tau\bar{\nu}$, $B_s \to D_s^{(*)}\tau\bar{\nu}$, as well as involving excited charm hadrons in the final state.

All existing measurements of $R(D^{(*)})$ rely heavily on large MC simulations to optimize selections, provide fit templates in discriminating kinematic observables, and to model resolution effects and acceptances. Both the τ and the charm hadrons have short lifetimes and decay near the interaction point and measurements rely on reconstruction of the ensuing decay cascades. To reconstruct the decay products, often complex phase space cuts and detector efficiency dependencies come into play, and the measurement of the full decay kinematics is impossible due to the presence of multiple neutrinos. In addition, depending on the final state, a significant downfeed with similar experimental signatures from misreconstructed excited charm hadron states can be present. Isolation of semitauonic decays from other background processes and the light-lepton final states, then requires precise predictions for the kinematics of the signal semitauonic decay. (Further complications arise from interference among the different spin states of the τ and among those of the charm hadron. Such effects have sometimes been neglected, treating the τ and charm hadron as stable particles, when simulations are corrected to account for more up-to-date hadronic models.) Often the limited size of the available simulated samples, required to account for all these effects, constitutes a dominant uncertainty of the measurements, see e.g. [10–12].

In the literature on the $R(D^{(*)})$ anomaly, it has become standard practice to reinterpret the experimental values of $R(D^{(*)})$ in terms of NP Wilson coefficients, even though all current ratio measurements were determined assuming the SM nature of semitauonic decays. However, NP couplings generically alter decay distributions and acceptances. Therefore, they modify the signal and possibly background MC templates used in the extraction, and thus affect the measured values of $R(D^{(*)})$. This may introduce biases in NP interpretations: preferred regions and best-fit points for the Wilson coefficients can be incorrect.

Consistent interpretations of the data with NP incorporated requires dedicated MC samples, ideally for each NP coupling value considered, which would permit directly fitting for the NP Wilson coefficients. This approach is sometimes referred to as 'forward-folding', and is naively a computationally prohibitively expensive endeavour. Such a program is further complicated because none of the MC generators current used by the experiments incorpo-

rate generic NP effects, nor do they include state-of-the-art treatments of hadronic matrix elements.

The Hammer software library, that provides a solution to these problems: A fast and efficient means to reweight large MC samples to any desired NP, or to any description of the hadronic matrix elements. Hammer makes use of efficient amplitude-level and tensorial calculation strategies, and is designed to interface with existing experimental analysis frameworks, providing detailed control over which NP or hadronic descriptions should be considered. The desired reweighting can be implemented either in the event weights or in histograms of experimentally reconstructed quantities. The only required MC input are the event-level truth-four-momenta of existing MC samples. Either the event weights and/or histogram predictions may be used, e.g., to generate likelihood functions for experimental fits. While Hammer has been designed primarily with $b \to c\tau\nu$ processes in mind – including not only $B \to D^{(*,**)}\ell\nu$, but also e.g. $\Lambda_b \to \Lambda_c\ell\nu$ or $B_c \to J/\psi\,\ell\nu$ – the general framework has been designed to be extendable to processes such as $c \to s\ell\nu$ or $b \to s\ell\ell$ among others.

II. DESIGN OVERVIEW

A. Reweighting

We consider an MC event sample, comprising a set of events indexed by I, with weights w_I and truth-level kinematics $\{q\}_I$. Reweighting this sample from an 'old' to a 'new' theory requires the truth-level computation of the ratio of the differential rates

$$r_I = \frac{d\Gamma_I^{\text{new}}/d\mathcal{PS}}{d\Gamma_I^{\text{old}}/d\mathcal{PS}}\,, \tag{2}$$

applied event-by-event via the mapping $w_I \mapsto r_I w_I$. The 'old' or 'input' or 'denominator' theory is typically the SM plus (where relevant) a hadronic model — that is, a form factor (FF) parametrization. (It may also be composed of pure phase space (PS) elements.) The 'new' or 'output' or 'numerator' theory may involve NP beyond the Standard Model, or a different hadronic model, or both.

Historically, the primary focus of the library is reweighting of $b \to c\ell\nu$ semileptonic processes, often in multistep cascades such as $B \to D^{(*,**)}(\to DY)\,\tau(\to X\nu)\bar{\nu}$. However, the library's computational structure is designed to be generalized beyond these processes, and we therefore frame the following discussion in general terms, before returning to the specific case of semileptonic decays.

B. New Physics generalization

The Hammer library is designed for the reweighting of processes under theories of the form

$$\mathcal{L} = \sum_{\alpha} c_\alpha \, \mathcal{O}_\alpha \, . \tag{3}$$

where $\mathcal{O}_\alpha$ are a basis of operators, and c_α, are SM or NP Wilson coefficients (defined at a fixed physical scale; mixing of the Wilson coefficients under RG evolution, if relevant, must be accounted for externally to the library). We specify in Table VI and the manual the conventions used for various $b \to c\ell\nu$ four-Fermi operators and other processes included in the library.

The corresponding process amplitudes may be expressed as linear combinations $c_\alpha \mathcal{A}_\alpha$. They may also be further expressed as a linear sum with respect to a basis of form factors, F_i, that encode the physics of hadronic transitions (if any).[1] In general, then, an amplitude may be written in the form

$$\mathcal{M}^{\{s\}}\big(\{q\}\big) = \sum_{\alpha,i} c_\alpha \, F_i\big(\{q\}\big) \, \mathcal{A}_{\alpha i}^{\{s\}}\big(\{q\}\big) \, , \tag{6}$$

in which $\{s\}$ are a set of external quantum numbers and $\{q\}$ the set of four-momenta.[2] The object $\mathcal{A}_{\alpha i}$ is an NP- and FF-generalized *amplitude tensor*. In the case of cascades, relevant for $B \to D^{(*,**)}(\to DY)\,\tau(\to X\nu)\bar{\nu}$ decays, the amplitude tensor may itself be the product of several subamplitudes, summed over several sets of internal quantum numbers. The corresponding polarized differential rate

$$\begin{aligned} \frac{d\Gamma^{\{s\}}}{d\mathcal{PS}} &= \sum_{\alpha,i,\beta,j} c_\alpha c_\beta^\dagger \, F_i F_j^\dagger\big(\{q\}\big) \, \mathcal{A}_{\alpha i}^{\{s\}} \mathcal{A}_{\beta j}^{\dagger\{s\}}\big(\{q\}\big) \, , \\ &= \sum_{\alpha,i,\beta,j} c_\alpha c_\beta^\dagger \, F_i F_j^\dagger\big(\{q\}\big) \, \mathcal{W}_{\alpha i \beta j} \, , \end{aligned} \tag{7}$$

in which the phase space differential form $d\mathcal{PS}$ includes on-shell δ-functions and geometric or combinatoric factors, as appropriate.

[1] In all $b \to c$ processes currently handled by Hammer– see Table III for a list – the form factors are functions of

$$q^2 = \big(p_{H_b} - p_{H_c}\big)^2 \, , \tag{4}$$

or equivalently functions of the dimensionless kinematic variable,

$$w = v \cdot v' = \frac{m_{H_b}^2 + m_{H_c}^2 - q^2}{2 m_{H_b} m_{H_c}} \, , \tag{5}$$

with four velocities $v = p_{H_b}/m_{H_b}$ and $v' = p_{H_c}/m_{H_c}$. For decays with multi-hadron final states, such as the $\tau \to n\pi$, $n \geq 3$, the form factors are also dependent on multiple invariant masses of the final state hadrons. Thus $b \to c\tau\nu$ processes with subsequent hadronic τ decays involve at least two separate sets of hadronic functions at the amplitude level.

[2] The momenta of an event passed to the library must all be specified in the same frame. The choice of frame is arbitrary.

The outer product of the amplitude tensor, defined as $\mathcal{W} \equiv \mathcal{A}\mathcal{A}^\dagger$, is a *weight tensor*. The object $\sum_{ij} F_i F_j^\dagger \mathcal{W}_{\alpha i \beta j}$ in Eq. (7) is independent of the Wilson coefficients: Once this object is computed for a specific $\{q\}$ – an event – it can be contracted with any choice of NP to generate an event weight. Similarly, on a patch of phase space Ω — e.g., the acceptance of a detector or a bin of a histogram — the marginal rate can now be written as

$$\Gamma_\Omega^{\{s\}} = \sum_{\alpha,\beta} c_\alpha c_\beta^\dagger \int_\Omega d\mathcal{PS} \sum_{ij} F_i F_j^\dagger(\{q\}) \mathcal{W}^{\{s\}}_{\alpha i \beta_j}(\{q\}) \,. \tag{8}$$

The Wilson coefficients factor out of the phase space integral, so that the integral itself generates a NP-generalized tensor. After it is computed once, it can be contracted with any choice of NP Wilson coefficients, c_α, thereafter.

The core of `Hammer`'s computational philosophy is based on the observation that this contraction is computationally much more efficient than the initial computation (and integration). Hence efficient reweighting is achieved by

- Computing NP (and/or FF, see below) generalized objects, and storing them;
- Contracting them thereafter for any given NP (and/or FF) choice to quickly generate a desired NP (and/or FF) weight.

C. Hadronic generalization

Similarly to the NP Wilson coefficients, it is often desirable to be able to generalize variation in the FF parameterization itself. For instance, one might contemplate variations along the error eigenbasis of a fit to the FF parameters, or FF parametrizations that are linearized with respect to a basis of parameters, such as the BGL FF parametrization [13–15] in $B \to D^{(*)}\ell\nu$. To this end, an FF parametrization with a parameter set $\{\mu\}$ can be linearized around a (best-fit) point, $\{\mu^0\}$ so that

$$F_i\big(\{q\};\{\mu\}\big) = F_i\big(\{q\},\{\mu^0\}\big) + \sum_a F'_{i,a}\big(\{q\},\{\mu^0\}\big)\, e_a \,, \tag{9}$$

where 'a' is one or more *variational indices* and e_a is the variation. In the language of the error eigenbasis case, $F'_{i,a}$ is the perturbation of F_i in the ath principal component e_a of the parametric fit covariance matrix.

Defining $\xi_a \equiv (1, e_a)$ and $\Phi_{i,a+1} \equiv (F_i, F'_{i,a})$, so that Eq. (9) becomes

$$\sum_a \xi_a \Phi_{i,a} = F_i + \sum_{a'} F'_{i,a'}\, e_{a'} \tag{10}$$

then the differential rate

$$\frac{d\Gamma^{\{s\}}}{d\mathcal{PS}} = \sum_{\alpha,a,\beta,b} c_\alpha c_\beta^\dagger \xi_a \xi_b^\dagger \mathcal{U}^{\{s\}}_{\alpha a \beta b} \,, \qquad \mathcal{U}^{\{s\}}_{\alpha a \beta b} \equiv \sum_{ij} \Phi_{i,a} \Phi^\dagger_{j,b}(\{q\}) \mathcal{W}^{\{s\}}_{\alpha i \beta_j}(\{q\}) \,, \tag{11}$$

with $\mathcal{U}$ an NP- and FF-generalized weight tensor. The ξ_a are independent of $\{q\}$ and factor out of any phase space integral just as the Wilson coefficients do. That is, an integral on any phase space patch,

$$\Gamma_{\Omega}^{\{s\}} = \sum_{\alpha,\beta,a,b} c_\alpha c_\beta^\dagger \xi_a \xi_b^\dagger \int_\Omega d\mathcal{PS}\, \mathcal{U}_{\alpha a \beta b}^{\{s\}} \,. \tag{12}$$

One may thus tensorialize the amplitude with respect to Wilson coefficients and/or FF linearized variations, to be contracted later with with NP or FF variation choices (the latter within the regime of validity of the FF linearization). Hereafter, the ξ_a are referred to as 'FF uncertainties' or 'FF eigenvectors' following the nominal fit covariance matrix example.

D. Rates

In certain use cases, it is also useful to compute and fold in an overall ratio of rates $\Gamma^{\rm old}/\Gamma^{\rm new}$, or the rates themselves, $\Gamma^{\rm new,old}$, may be required. For example, if the MC sample has been initially generated with a fixed overall branching ratio, $\mathcal{B}_{\rm new}$, one might wish to enforce this constraint via an additional multiplicative factor $\mathcal{B}_{\rm old}/\mathcal{B}_{\rm new}$.

The different components computed by Hammer are then:

(i) The NP- and/or FF-generalized tensor for $(d\Gamma_I^{\rm new}/d\mathcal{PS})/(d\Gamma_I^{\rm old}/d\mathcal{PS})$, via Eq. (11), noting that the denominator carries no free NP or FF variational index. (The ratio r_I is then itself generally at least a rank-2 tensor.);

(ii) The NP- and/or FF-generalized *rate tensors* $\Gamma^{\rm old,\ new}$, which need be computed only once for an entire sample. (These rates require integration over the phase space, which is achieved by a dedicated multidimensional Gaussian quadrature integrator.)

E. Primary code functionalities

The calculational core of Hammer computes the NP or FF generalized tensors event-by-event for any process known to the library (see Tab. III for a list), and as specified by initialization choices (more detail is provided in Sec. II F) and specified form factor parametrizations. This core is supplemented by a wide array of functionalities to permit manipulation the resulting NP- and FF-generalized weight tensors as needed. This may include binning — equivalent to integrating on a phase space patch — the weight tensors into a histogram of any desired reconstructed observables, and/or it may include folding of detector simulation smearings, etc. Such histograms have NP- and FF-generalized tensors as bin entries, and we therefore call them *generalized* or *tensor* histograms. Once such NP- and FF-generalized tensor objects are computed and stored, contraction with NP or FF eigenvector choices permits the library to efficiently generate actual event weights or histogram bin weights for any theory of interest.

The architecture of Hammer is designed around several primary functionalities:

(i) Provide an interface to determine which processes are to be reweighed, and which (possibly multiple) schemes for form factor parametrizations are to be used. This includes handling for (sub)processes that were generated as pure phase space.

(ii) Parse events into cascades of amplitudes known to the library, and compute their corresponding NP- and/or FF-generalized amplitude or weight tensor, as well as the respective rate tensors, as needed.

(iii) Provide an interface to generate histograms (of arbitrary dimension), and bin the event weight tensors — i.e., $r_I w_I$, as in Eq. (2) — into these histograms, as instructed. This includes functionality for weight-squared statistical errors, functionality for generation of ROOT histograms, as well as extensive internal architecture for efficient memory usage.

(iv) Efficiently contract generalized weight tensors or bin entries against specific FF variational or NP choices, to generate an event or bin weight. This includes extensive internal architecture to balance speed versus memory requirements.

(v) Provide interface to save and reload amplitude or weight tensors or generalized histograms, to permit quick reprocessing into weights from precomputed or 'initialized' tensor objects.

Examples of the implementation of these functionalities is shown in extensive examples provided with the source code.

F. 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 *analysis* 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 Wilson coefficients for the input MC via `setWilsonCoefficients`.

(viii) (Optional) Fix Wilson coefficient (Wilson coefficient and/or FF uncertainty) choice to special choices in weight calculations (histogram binnings), via `specializeWCInWeights` (`specializeWCInHistogram` and/or `specializeFInHistogram`).

(ix) 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 programmatic means – and specify the corresponding histogram bins to be filled via `fillEventHistogram`.

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

  (d) (Optional) Save the weight tensors for each event, with `saveEventWeights` to a buffer.

  (e) (Optional) Save the rate tensors, with `saveRates` to a buffer.

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

(xi) (Optional) Save an autogenerated `bibTeX` list of references used in the run with `saveReferences`.

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) Load or merge the run header — 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) Load or merge saved histograms with `loadHistograms`, and/or generate desired histograms with `getHistogram(s)`. NP choices are implemented with `setWilsonCoefficients`.

(iv) (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`.

III. THE HAMMER FORGE

In the following we describe various core parts of the `Hammer` Application Programming Interface (API). This includes a detailed explanation of information handling and rules enforced by the computational core in following user specifications, assembling amplitudes, or returning histograms or weights, among other functionalities. The library itself is implemented in `C++`, along with a `Python3` wrapper of the API, that uses idenitical syntax.

We will consider here the `C++` interface only; the discussion is ordered by scope, rather than the typical code flow. The library provides four core classes in its user interface: the `Hammer` class itself; the `Process` and `Particle` classes, used to create events; and the `IOBuffer` class used for saving and loading precomputed objects. Internal computational classes include `Amplitude`, `FormFactor` and `Rate` classes, that encode the physics of processes known to the library. A schematic of the architecture of `Hammer` is shown in Fig. 1.

A. From the process tree to an amplitude tensor

A typical decay cascade is contained in the library by the `Process` class; an event may contain multiple `Process` instances as e.g., is the case for a signal plus tag B-$\bar{B}$ pair. Each cascade may be simply represented in graphical terms as a 'process tree', as shown in Fig. 2: Each decay vertex is labelled by its local parent particle, connected to subsequent daughter decays by an edge (i.e. a line, or formally, a propagator). Each particle in the cascade is itself assigned an index, and then decay vertex is represented as a map from a parent index, to the indices of all its daughters.

`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

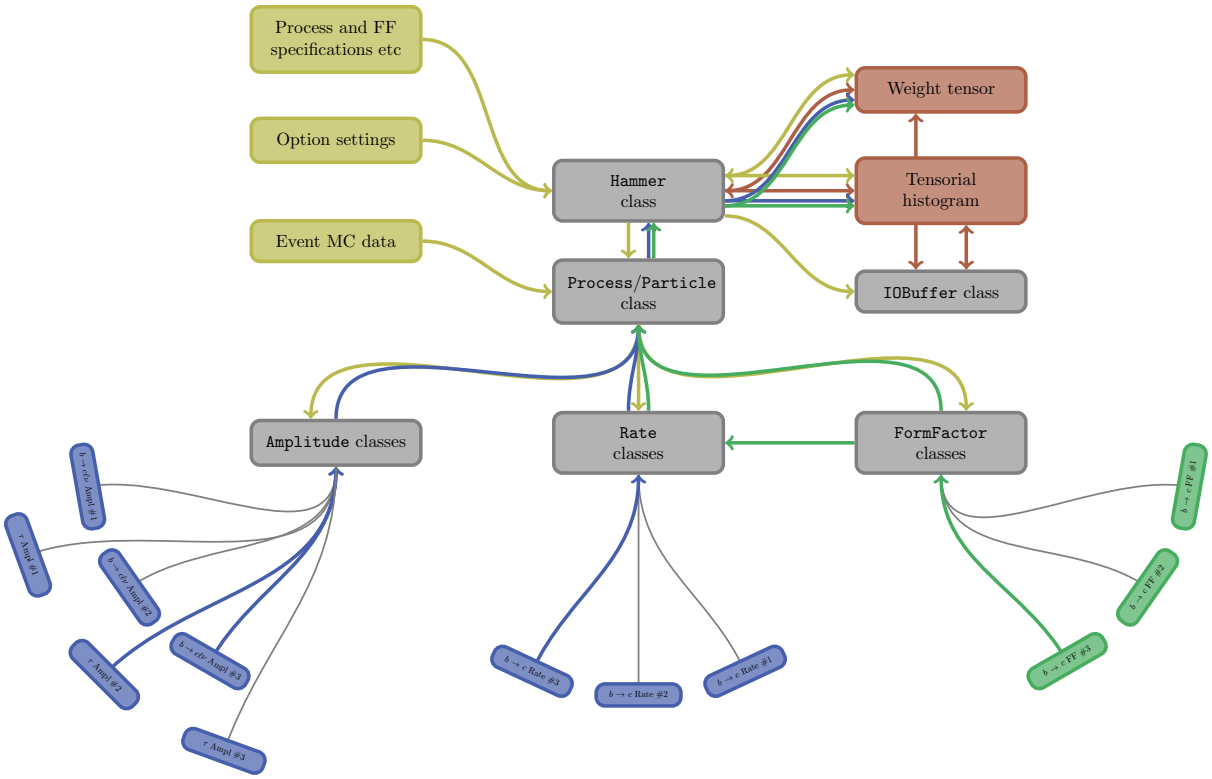


FIG. 1. Schematic architecture of Hammer. The flow of user specified choices or event data is shown by yellow arrows. Blue (green) arrows denote the flow of calculational information, in particular amplitude, weight or rate (form factor) tensors. Red arrows highlight the flow of Hammer output, which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

vertex. In the case of Fig. 2, the first two vertices of the cascade may be built explicitly as follows:

```
Process proc;
size_t idx0 = proc.addParticle(Particle{{E_0, px_0, py_0, pz_0}, pdg_0});
size_t idx1 = proc.addParticle(Particle{{E_1, px_1, py_1, pz_1}, pdg_1});
size_t idx2 = proc.addParticle(Particle{{E_2, px_2, py_2, pz_2}, pdg_2});
size_t idx3 = proc.addParticle(Particle{{E_3, px_3, py_3, pz_3}, pdg_3});
size_t idx7 = proc.addParticle(Particle{{E_7, px_7, py_7, pz_7}, pdg_7});
size_t idx8 = proc.addParticle(Particle{{E_8, px_8, py_8, pz_8}, pdg_8});

proc.addVertex(idx0, {idx1,idx2,idx3});
proc.addVertex(idx2, {idx7,idx8});
```

and so on. Particles and vertices need not be added in order; helper functions are provided in the code examples for automatically parsing HepMC files.

From the filled process tree, Hammer determines several hashes or sets of hashes, that encode

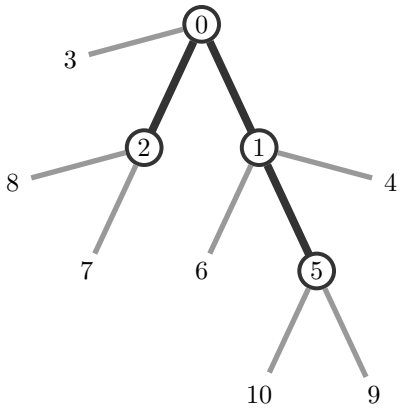


FIG. 2. Example process tree for a decay cascade involving 10 particles (numbers), 4 vertices (circles) and 3 edges (dark lines).

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 amplitudes and form factor parametrizations

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. See the option card (OptDefaults.pdf) for a full list of the settable form factor parameters and switches, and their default values.

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 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

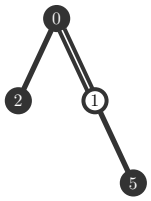
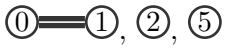
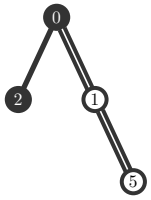
Known Amplitudes	Evaluated Amplitudes
	
	

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

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 O).

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

```
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

Process	FF parametrizations
$B \to D^{(*)}\ell\nu$	ISGW2* [16, 17], BGL*‡ [13–15], CLN*‡ [18], BLPR‡ [19], BLPRXP‡ [20]
$B \to (D^* \to D\pi)\ell\nu$	ISGW2*, BGL*‡, CLN*‡, BLPR‡, BLPRXP‡
$B \to (D^* \to D\gamma)\ell\nu$	ISGW2*, BGL*‡, CLN*‡, BLPR‡, BLPRXP‡
$B \to D_0^*\ell\nu$	ISGW2*, LLSW* [21, 22], BLR‡ [23, 24]
$B \to D_1^*\ell\nu$	ISGW2*, LLSW*, BLR‡
$B \to D_1\ell\nu$	ISGW2*, LLSW*, BLR‡
$B \to D_2^*\ell\nu$	ISGW2*, LLSW*, BLR‡
$B \to (\rho \to \pi\pi)\ell\nu$	ISGW2*, BSZ‡ [25]
$B \to (\omega \to \pi\pi\pi)\ell\nu$	ISGW2*, BSZ‡
$\Lambda_b \to \Lambda_c\ell\nu$	PCR* [26], BLRS‡ [27, 28]
$\Lambda_b \to \Lambda_c^*\ell\nu$	PCR*, LSPR‡[29, 30]
$B_c \to (J/\psi \to \ell\ell)\ell\nu$	Kiselev* [31], EFG* [32], BGL*‡ [33], ...
$B \to \pi\ell\nu$	ISGW2*, BCL*‡ [34], GKvD [35]
$\tau \to \pi\nu$	—
$\tau \to \ell\nu\nu$	—
$\tau \to 3\pi\nu$	RCT* [36–38]
$D_1 \to (D^* \to D\pi/\gamma)\pi$	PW
$D_2^* \to (D^* \to D\pi/\gamma)\pi$	PW
$D_2^* \to D\pi$	PW
Planned for future release	
$B_{(c)} \to \ell\nu$	MSbar
$\tau \to 4\pi\nu$	RCT*
$\tau \to (\rho \to \pi\pi)\nu$	—

TABLE III. Presently implemented amplitudes in Hammer and corresponding form factor parametrizations. SM-only parametrizations are indicated by a $*$ superscript. Form factor parameterizations that include linearized variations are denoted with a ‡ superscript. These are named in the library by adding a "Var" suffix, e.g. "BGLVar".

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 ]
```

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' to be used in reweighting. A form factor scheme is a set of FF parameterization choices for each hadronic transition involving form factors, 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. The hadronic strings are charge sensitive, so different FFs for charged and neutral processes can be set, e.g. via an entry {"B+D", "BGL"} versus {"B0D", "CLN"}, 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 }
```

Symbol	Particle(s)
D	D^+, D^-, D^0, $\bar{D}^0$
D*	D^{*0}, D^{*-}, D^{*+}, $\bar{D}^{*0}$
Lc	Λ_c^+, Λ_c^-
B	B^0, B^-, B^+, $\bar{B}^0$
Lb	Λ_b^0, $\bar{\Lambda}_b^0$
K	K^+, K^-, K_L^0, K_S^0
Pi	π^0, π^+, π^-
D**0*	D_0^{*0}, D_0^{*-}, D_0^{*+}, $\bar{D}_0^{*0}$
D**1*	D_1^*, D_1^{*-}, D_1^{*+}, $\bar{D}_1^{*0}$
D**1	D_1^0, D_1^-, D_1^+, $\bar{D}_1^0$
D**2*	D_2^{*0} D_2^{*-}, D_0^{*+}, $\bar{D}_0^{*0}$
Lc*2595	$\Lambda_c^{*+}(2595)$, $\Lambda_c^{*-}(2595)$
Lc*2625	$\Lambda_c^{*+}(2625)$, $\Lambda_c^{*-}(2625)$
E	e^+, e^-
Mu	μ^-, μ^+
Tau	τ^-, τ^+
Nu	ν_e, $\bar{\nu}_e$, ν_μ, $\bar{\nu}_\mu$, ν_τ, $\bar{\nu}_\tau$
Ell	μ^-, μ^+, e^-, e^+
W	W^+, W^-
Gamma	γ
Tau+	τ^+
E+	e^+
Mu+	μ^+
K+	K^+
K0S	K_S^0
K0L	K_L^0
B0	B^0
B+	B^+
B0bar	$\bar{B}^0$
B+-	B^+, B^-
Babar	B^0, $\bar{B}^0$
D0	D^0
D+	D^+
D0bar	$\bar{D}^0$
D+-	D^+, D^-
Dabar	D^0, $\bar{D}^0$

Symbol	Particle(s)
D*0	D^{*0}
D*+	D^{*+}
D*-	D^{*-}
D*0bar	$\bar{D}^{*0}$
D*+-	D^{*+}, D^{*-}
D*abar	D^{*0}, $\bar{D}^{*0}$
Lc+	Λ_c^+
Lc*2595+	$\Lambda_c^{*+}(2595)$
Lc*2625+	$\Lambda_c^{*+}(2625)$
Lb0	Λ_b^0
Lb0bar	$\bar{\Lambda}_b^0$
Pi0	π^0
Pi+	π^+
Nut	ν_τ
Nutbar	$\bar{\nu}_\tau$
Num	ν_μ
Numbar	$\bar{\nu}_\mu$
Nue	ν_e
Nuebar	$\bar{\nu}_e$
W+	W^+
D**0*0	D_0^{*0}
D**0*+	D_0^{*+}
D**0*0bar	$\bar{D}_0^{*0}$
D**0*+-	D_0^{*+}, D_0^{*-}
D**0*abar	D_0^{*0}, $\bar{D}_0^{*0}$
D**1*0	D_1^{*0}
D**1*+	D_1^{*+}
D**1*0bar	$\bar{D}_1^{*0}$
D**1*+-	D_1^{*+}, D_1^{*-}
D**1*abar	D_1^{*0}, $\bar{D}_1^{*0}$
D**10	D_1^0
D**1+	D_1^+
D**10bar	$\bar{D}_1^0$
D**1+-	D_1^+, D_1^-
D**1abar	D_1^0, $\bar{D}_1^0$
D**2*0	D_2^{*0}
D**2*+	D_2^{*+}
D**2*0bar	$\bar{D}_2^{*0}$
D**2*+-	D_2^{*+}, D_2^{*-}
D**2*abar	D_2^{*0}, $\bar{D}_2^{*0}$

TABLE IV. List of currently available particle specifications and corresponding particles. For each '...+' name, there is a corresponding '...-'.

```
   Scheme2: { BD: BGL, BD*: CLN }
Denominator: { BD: ISGW2, BD*: ISGW2 }
```

E. Form factor settings

FF parametrization default settings are fixed inside the FF classes themselves. 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. This can be done before or after invocation of `initRun`. Note that the YAML key for the relevant FF class matches the format of the *class prefix*, with '`to`' inserted in the hadronic transition, producing an `"XtoY"` form. E.g. `BtoDBGL`, rather than `BDBGL`.[3]

See the option card (`OptDefaults.pdf`) for a full list of the settable form factor parameters and switches, and their default values.

F. 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}");
```

G. 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 P below.)

[3] This notation is intended to make it clear we are identifying settings for a particular class – the $B \to D$ BGL class – and not a process. It further ensures syntactic distinction between a hadronic string representation, which can take a charge assignment like `"B0D+"`, and a class prefix for a $B \to D$ FF class like `"BtoD"`, which does not.

H. 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; and calculating the total rate for the vertex (this is done only once per run per unique vertex and per FF scheme). The amplitude tensors and form factors 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 specified to be filled; see Sec. III M 2), the amplitudes and weights 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.

Internally, `processEvent` proceeds by two separate steps: Calculating the process amplitudes and weight; and then filling histograms (if any). Either of these steps can be disabled at anytime by the options settings `ham.setOptions("Hammer: {CalcProcesses: false}")` and `ham.setOptions("Hammer: {CalcHistograms: false}")`, respectively, and similarly re-enabled at

any time. Alternatively, `processEvent` can accept an optional enum parameter of type `PAction` as input. The default behavior of computing both weights and filling histograms is `PAction::ALL`, but can be modified to weights-only by passing `PAction::WEIGHTS` or to histograms-only with `PAction::HISTOGRAMS`. In the latter case the weights must have already been computed in an earlier call. These might be useful when combined with other error handling inputs.

MC event samples with very large numbers of events but low numerical precision can lead to rare events with 'impossible' kinematics. For example, in an $X \to YZ$ vertex, if Y and Z have very small angular separation, numerical noise can lead to helicity angle cosines, when expressed in terms of kinematic invariants, that fluctuate > 1. This can lead to a `NaN` amplitude and event weight. The option `ham.setOptions("ProcessCalc: {CheckForNaNs: true}")` allows for explicit checking of `NaN` at the process amplitude calculational step, throwing an error that may be caught upstream as desired (e.g. within a conditional on whether `processEvent` should fill histograms).

I. 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`.

J. Setting Wilson Coefficients

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 H.) 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 argument is an enum `WTerm` value, that declares whether the evaluation should be applied to the numerator and/or denominator (numerator by default). The enum `WTerm` may take values `{COMMON, NUMERATOR, DENOMINATOR}` 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"}.
```

It is important to note that the `setWilsonCoefficients` method, when taking a `map`, produces *incremental* settings changes. I.e. the sequential invocations

```
ham.setWilsonCoefficients("BtoCTauNu",
        {{"S_qLlL", 1.}, {"T_qLlL",0.25}});
ham.setWilsonCoefficients("BtoCTauNu", {{"S_qLlL", 0.5}});
```

will result in `S_qLlL` $= 0.5$ and `T_qLlL` $= 0.25$, since the latter was not affected by the second call. The method `resetWilsonCoefficients` takes the WC type – e.g. `"BtoCTauNu"`– and resets the corresponding WCs to the default SM.

K. Setting FF eigenvectors

As mentioned in Sec II C, certain form factor classes (typically, those with names ending in "`Var`") incorporate linearized variation of the FF parametrization, with additional variational indices in the form factor tensors, in the sense defined by eq. (9). 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)vector, 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 examples of definitions and conventions of currently implemented FF variational classes. 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).

The FF classes with linearized variations permit the matrix of eigenvectors of the fit covariance – the eigenspace matrix, that defines the basis of variations – to be set as an option through `setOptions`. (The special choice that this eigenspace matrix is the identity typically corresponds to the choice that each eigendirection is actually just motion in the

underlying linearized space of FF parameters.) These classes similarly permit the naming scheme for the basis of variations to be adjusted by changing the vector of names using the method `renameFFEigenvectors`, in order to accomodate different conventions for this matrix. For example, if the eigenspace matrix is the identity, it is clearer to label the basis of variations with respect to the parameter names, e.g. `"delta_a1"` for variation in the a_1 $B \to D^*$ BGL parameter and so forth (see Sec. V E). However, if the eigenspace matrix is instead actual eigenvectors, one might prefer `"delta_e1"` and so on.

For example, if the $X \to Y$ FF class `DemoVar` has default basis `{delta_a, delta_b, delta_c, delta_d}`, this may be changed via

```
ham.renameFFEigenvectors("XtoY", "DemoVar", {"delta_e1", "delta_e2", "delta_e3",
    "delta_e4"});
```

This can be done before or after invocation of `initRun`. A warning will be thrown if the renaming list is longer than the basis defined in the class. Passing a list of k names that is shorter than defined in the class will rename only the first k; an empty string at the i-th position leaves the name at the i-th position unchanged. For example, the list of names `{"", "delta_e2"}` in the above example would leave `"delta_a"`, change `"delta_b"`, then leave all remaining names unchanged.

As for WCs, the `setFFEigenvectors` method, when taking a `map`, produces *incremental* settings changes. The method `resetFFEigenvectors` takes the name of the hadronic process in `"XtoY"` form and the name of the FF parametrization, and resets the corresponding FF eigenvectors to zero.

L. Specialization of Wilson Coefficients

One may wish to fix *a priori* all WCs of a specific type at the time of tensor weight computation, in order to reduce space or reweighting times. This acheived with the methods `specializeWCInWeights` that takes the arguments required by `setWilsonCoefficients`. An example usage

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

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

Specialization is not reversible once a weight is computed in an initialization run by `processEvent`. However a definition reset method `resetSpecializeWCInWeights` is also provided, which may e.g. be invoked before a subsequent initialization run to turn off the WC specialization.

M. 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 contracted histogram contains elements that are `BinContents` structs, with members `sumWi`, `sumWi2` and `n` for sum of weights, sum of squared weights and number of events in the bin, respectively.

1. *Adding*

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* and in turn for each FF scheme name specified by `addFFScheme`. Here an event ID is a `set` of process IDs for all processes included in the event.[4] 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). With reference to the above `addFFScheme` example in Sec. III D, this histogram set contains one histogram for each combination of either `"Scheme1"` or `"Scheme2"` with each unique $B \to D$ decay cascade. Alternatively, for non-uniform bins

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

which creates a 3×2 histogram, 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"`.

[4] Because of histogram compression functionality discussed in Sec. III M 3 below, histograms are in practice indexed by a *event ID group*, which is a set of event IDs: Without compression each event ID group is just a trivial single element set containing the event ID of the histogram.

2. Filling

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 histogram. When the under/overflow bool is set to `false`, events outside the bin ranges are ignored by `fillEventHistogram`.

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 M 5). This method should invoked before `initRun`. The`"Total Sum of Weights"` histogram, if it has been created, is automatically filled by `processEvent`.

3. Compression

In many use cases, the entire histogram set is not required, but rather its direct sum. Computing and storing only the latter compressed form 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 compressed histogram. For instance,

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

This method should invoked before `initRun`. When invoked, each compressed histogram in the histogram set is then indexed by non-trivial event ID groups, containing the event IDs of all the histograms that were collapsed into it.

4. Retrieval

Once all events or histograms have been processed (or reloaded from a file, see Sec. IV) the user may retrieve a specific histogram via 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 contract the bin weights with the specified NP Wilson coefficients (and FF eigenvectors, if any) for each histogram in the `"q2VsEmu"` histogram set populated for `"Scheme2"`, and then combines them together into a single final histogram. This contracted histogram output `histo` is a (row-major) flattened vector of `BinContents` structs. 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 with FF scheme `"Scheme2"`.

5. *Specialization*

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`.

Specialization is not reversible once a histogram is filled in an initialization run. However a definition reset method `resetSpecializationInHistogram` is also provided, which may e.g. be invoked before a subsequent initialization run to reset the histogram definition to the default and turn off its specializations.

6. *Uncertainties*

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.

7. *Projection*

On occasion it may be useful to project a pre-computed n-dimensional histogram onto a lower dimensional one. This can be achieved via the method createProjectedHistogram, which takes the name of the original n-dimensional histogram, the name of the new histogram to be created, and a set of the index positions to be summed over or collapsed. For instance, for a 3-dimensional histogram "q2VsEmuVsM2miss" with dimensions q^2, E_μ and m^2_{miss}, one may integrate over the E_μ and m^2_{miss} dimensions to create a 1-dimension q^2 histogram via

```
ham.createProjectedHistogram("q2VsEmuVsM2miss", "justq2", {1,2});
```

in which the new histogram, named "justq2", inherits the underlying structure – the histogram set – of the original histogram.

N. 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 evaluated as 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

$$\frac{1}{\prod_k |\{s_k\}|} \sum_{s_i, r_j} \left| \mathcal{M}_{s_1,\ldots,s_n;r_1,\ldots,r_m} \right|^2 = 1 \times (m^{6-2n}) \,, \tag{13}$$

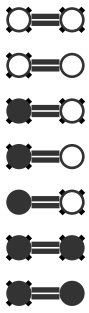
where s_i (r_i) are incoming (outgoing) quantum numbers, $|\{s_k\}|$ is the number of states of s_k, m is the mass of the parent particle in the vertex, and n the number of daughters. I.e., the squared matrix element averaged over initial states and summed over final states is set to unity times a factor 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.

O. PHOTOS

Typical MC samples include collinear radiative corrections, incoherently appended to the relevant vertices by the PHOTOS algorithm [39], 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.

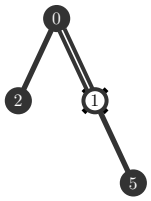
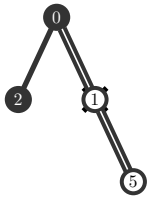
Known Amplitudes	Evaluated Amplitudes
	⓪, ②, ⑤
	⓪, ①═⑤, ②

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

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.)

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 [39], 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, γ_{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 γ_{rad}, collectively called 'Y', and all their descendants, the 'Y subtree'. The radiative vertex is thus written $P \to \text{ch} + Y + \gamma_{\text{rad}}$.

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

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

$$\beta\gamma_{\text{ch}} = \frac{E_{\text{ch}}|\boldsymbol{p}^*| - E^*_{\text{ch}}|\boldsymbol{p}_{\text{ch}}|}{m^2_{\text{ch}}}\,, \qquad \beta\gamma_Y = \frac{E_Y|\boldsymbol{p}^*| - E^*_Y|\boldsymbol{p}_Y|}{m^2_Y}\,, \tag{14}$$

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

$$E^*_{\mathrm{ch}} = \frac{m_P^2 - m_Y^2 + m_{\mathrm{ch}}^2}{2m_P}\,, \qquad E^*_Y = \frac{m_P^2 - m_{\mathrm{ch}}^2 + m_Y^2}{2m_P}\,, \qquad |\boldsymbol{p}^*| = \frac{m_P}{2}\lambda^{1/2}\left[\frac{m_Y}{m_P}, \frac{m_{\mathrm{ch}}}{m_P}\right], \tag{15}$$

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

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

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

P. 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 G). 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 N) then following the PS

definition (13) the returned rate for that vertex is the phase space rate

$$\Gamma_n^{\mathcal{PS}} = \frac{1}{2m} \int m^{6-2n} d\mathcal{PS}_n \,. \tag{16}$$

The rates for vertices whose amplitudes have no form factor – they are not required to be specified in a FF scheme – are automatically assigned to each scheme name relevant for the decay process to which they belong. For example in a $B \to D(\tau \to \mu\nu\nu)\nu$ decay with schemes `"Scheme1"` and `"Scheme2"`, the $\tau \to \mu\nu\nu$ partial width can be retrieved via `ham.getRate("Tau+Mu+NuNu", "Scheme1")` or `ham.getRate("Tau+Mu+NuNu", "Scheme2")`.

Q. Multithreading

The library has the ability to perform lock-free parallelization of the `getHistogram(s)` and `getWeight` evaluations. This requires use of the thread local methods `setWilsonCoefficientsLocal` and `setFFEigenvectorsLocal` to set the desired WC or FF uncertainties.

The `...Local` methods take the same syntax as their global versions `setWilsonCoefficients` and `setFFEigenvectors`, but with different behaviour: They do not set the values incrementally from the current settings, but always increment from the SM and zero FF uncertainties, respectively. Global values of the WCs or FF variations are unaffected by the `...Local` methods, but the global `set...` methods should not be used in a multithreaded run.

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

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

A. Saving

1. *Headers, Events and Rates*

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'`, `HISTOGRAM_DEFINITION = 'd'`, and `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.) Any combination of save methods may be invoked, in any order.

The method `saveEventWeights` saves the event weights of the currently initialized and processed event (there may be multiple weights 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.

2. *Histograms*

The method `saveHistogram`, when taking only a histogram name as an argument, saves the entire specified histogram set (each histogram in an histogram set occupies an individual buffer record). 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 M 3). Invocation of `saveHistogram` automatically also saves an additional separate buffer record for the histogram definition, immediately preceding the histogram record itself.

The `saveHistogram` method may optionally take additional arguments – either an FF scheme name or an event ID group – in order to save only part of an entire histogram set, if e.g. space or file sizes are too large for an entire histogram set. For instance,

```
outFile << ham.saveHistogram("q2VsEmu", "Scheme2");
```

saves only the those histograms in the histogram set computed for `Scheme2` and not for any other schemes that may have been declared. If event ID groups are known, one may instead save histograms for one group, `eventIDgroup`, via

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

Attempting to save a histogram that does not exist will result in an exception.

3. *ROOT*

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. (Re)loading

Buffer records may be loaded from a declared `ifstream` infile into an `IOBuffer` via an `istream` operator, using `load...` methods with the same nomenclature as the `save...` methods. 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).

It is the responsibility of the user to curate the logic and order under which a buffer is saved and then read. For example, if a block of histograms have been saved before a set of rate records, then

```
while(buf.kind != Hammer::RecordType::RATE) {
  if(buf.kind == Hammer::RecordType::HISTOGRAM) {
    ham.loadHistogram(buf);
  }
  if(buf.kind == Hammer::RecordType::HISTOGRAM_DEFINITION){
    ham.loadHistogramDefinition(buf);
  }
  inFile >> buf;
}
```

would read through the buffer, with the method `Hammer::loadHistogram` loading all the histograms, and `Hammer::loadHistogramDefinition` all the histogram definitions, that are found before reaching the block of saved rates. (One could instead have used `while(buf.kind != Hammer::RecordType::UNDEFINED)` to simply read through the entire buffer.)

Once an object is loaded, it behaves just as the originally computed instance. Thus one may invoke `getHistogram` for a reloaded histogram as described in Sec. III M 4. (The method `removeHistogram` takes a histogram name and permits deletion of that histogram from the instance.)

Event weights can be reloaded via `loadEventWeights`. This permits recreating the original event loop provided `initEvent` and `processEvent` are called appropriately. For example, on a block of saved event records

```
while(buf.kind == Hammer::RecordType::EVENT) {
  ham.initEvent();
  ham.loadEventWeights(buf);
```

```
  double q2 = ...; //Calculate q^2 from known kinematic event information
  ham.fillEventHistogram("Q2", {q2});
  ham.processEvent();
  inFile >> buf;
}
```

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

The method `loadRates` behaves similarly to `loadHistogram`. 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 in memory (`true`), or overwrite them (`false`, default).

First, `loadRunHeader` permits merging of two sets of header settings into their union, with errors thrown for matching settings with non-matching values. When merging, the first invocation of `loadRunHeader` should be called with the merge bool set to `false`, and subsequent invocations set to `true`. (The other load methods may be uniformly called with the merge bool set to `true`.)

Merging of histograms occurs if two histograms are loaded with a matching name. This merging is *additive* for histograms in each histogram set with the same event ID group and FF scheme, and otherwise results in the new unique histograms being *appended* to the existing histogram set. (If one wishes instead to overwrite a histogram one may instead first

invoke `removeHistogram`, and then reload the desired components of the 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.

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). In the case of merging rates, errors are thrown if a form factor scheme by the same name already exists for the same decay and the rate tensors do not match.

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 $V_{cb}\eta_{\rm EW}$. 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.

These conventions correspond to the conventions of Refs. [40] via

$$\begin{aligned} \chi^V_L &= \alpha^{V*}_L\,, & \chi^V_R &= \alpha^{V*}_R\,, \\ \chi^S_R &= -\alpha^{S*}_L\,, & \chi^S_L &= -\alpha^{S*}_R\,, \\ \chi^T_R &= -\alpha^{T*}_L\,, & \chi^T_L &= -\alpha^{T*}_R\,, \\ \lambda^{V,S,T}_L &= \beta^{V,S,T*}_L\,, & \lambda^{V,S,T}_R &= \beta^{V,S,T*}_R\,. \end{aligned} \tag{17}$$

Current	WC Tag	WC	4-Fermi/$(i2\sqrt{2}\,V_{cb}G_F)$
SM	SM	1	$[\bar{c}\gamma^\mu P_L b]\,[\bar{\ell}\gamma_\mu P_L \nu]$
Vector	V_qLlL	$\chi_L^V \lambda_L^V$	$[\bar{c}\chi_L^V\gamma^\mu P_L b]\,[\bar{\ell}\lambda_L^V\gamma_\mu P_L \nu]$
	V_qRlL	$\chi_R^V \lambda_L^V$	$[\bar{c}\chi_R^V\gamma^\mu P_R b]\,[\bar{\ell}\lambda_L^V\gamma_\mu P_L \nu]$
	V_qLlR	$\chi_L^V \lambda_R^V$	$[\bar{c}\chi_L^V\gamma^\mu P_L b]\,[\bar{\ell}\lambda_R^V\gamma_\mu P_R \nu]$
	V_qRlR	$\chi_R^V \lambda_R^V$	$[\bar{c}\chi_R^V\gamma^\mu P_R b]\,[\bar{\ell}\lambda_R^V\gamma_\mu P_R \nu]$
Scalar	S_qLlL	$\chi_L^S \lambda_L^S$	$[\bar{c}\chi_L^S P_L b]\,[\bar{\ell}\lambda_L^S P_L \nu]$
	S_qRlL	$\chi_R^S \lambda_L^S$	$[\bar{c}\chi_R^S P_R b]\,[\bar{\ell}\lambda_L^S P_L \nu]$
	S_qLlR	$\chi_L^S \lambda_R^S$	$[\bar{c}\chi_L^S P_L b]\,[\bar{\ell}\lambda_R^S P_R \nu]$
	S_qRlR	$\chi_R^S \lambda_R^S$	$[\bar{c}\chi_R^S P_R b]\,[\bar{\ell}\lambda_R^S P_R \nu]$
Tensor	T_qLlL	$\chi_L^T \lambda_L^T$	$[\bar{c}\,\chi_L^T\sigma^{\mu\nu} P_L b]\,[\bar{\ell}\,\lambda_L^T\sigma_{\mu\nu} P_L \nu]$
	T_qRlR	$\chi_R^T \lambda_R^T$	$[\bar{c}\,\chi_R^T\sigma^{\mu\nu} P_R b]\,[\bar{\ell}\,\lambda_R^T\sigma_{\mu\nu} P_R \nu]$

TABLE VI. NP operator basis, and coupling conventions.

All internal Hammer calculations are done in the $\alpha_j^i\beta_l^k$ basis of Ref. [40], 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 $\chi_j^i\lambda_l^k$ WC basis. In the conventions of Ref. [23], $\chi = \tilde{\alpha}$, 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

$$\mathrm{Tr}[\gamma^\mu\gamma^\nu\gamma^\sigma\gamma^\rho\gamma^5] = -4i\epsilon^{\mu\nu\rho\sigma}\,, \qquad \epsilon^{0123} = +1\,. \tag{18}$$

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

$$\mathrm{FF}_D = \Big\{ f_S,\, f_0,\, f_+,\, f_T \Big\}\,, \tag{19}$$

which are defined via

$$\big\langle D \big|\, \bar{c}\, b \,\big|\overline{B}\big\rangle \equiv f_S\,, \tag{20a}$$

$$\big\langle D \big|\, \bar{c}\gamma^\mu b \,\big|\overline{B}\big\rangle \equiv f_+(p_B + p_D)^\mu + [f_0 - f_+]\,\frac{m_B^2 - m_D^2}{q^2}\,q^\mu\,, \tag{20b}$$

$$\big\langle D \big|\, \bar{c}\sigma^{\mu\nu} b \,\big|\overline{B}\big\rangle \equiv i f_T \Big[(p_B + p_D)^\mu q^\nu - (p_B + p_D)^\nu q^\mu\Big]\,. \tag{20c}$$

These definitions map to the conventional dimensionless form factor set h_S, h_+, h_-, h_T, as defined in e.g. Ref. [19], via

$$f_S = \sqrt{r_D}(w+1) m_B h_S\,, \tag{21a}$$

$$f_0 = \frac{\sqrt{r_D}}{r_D^2 - 1}\big[(r_D+1)(w-1)h_- + (r_D-1)(w+1)h_+\big] \tag{21b}$$

$$f_+ = \frac{(r_D-1)h_- + (r_D+1)h_+}{2\sqrt{r_D}}\,, \tag{21c}$$

$$f_T = \frac{h_\mathrm{T}}{2\sqrt{r_D} m_B}\,, \tag{21d}$$

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. (21).

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

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

$$\mathrm{FF}_{D^*} = \Big\{ a_0,\, f,\, g,\, a_-,\, a_+,\, a_{T_0},\, a_{T_-},\, a_{T_+} \Big\}\,, \tag{22}$$

which are defined via

$$\big\langle D^* \big|\, \bar{c}\gamma^5 b \,\big|\overline{B}\big\rangle \equiv a_0\, \varepsilon^* \cdot p_B\ , \tag{23a}$$

$$\big\langle D^* \big|\, \bar{c}\gamma^\mu b \,\big|\overline{B}\big\rangle \equiv -ig\,\epsilon^{\mu\nu\rho\sigma}\,\varepsilon^*_\nu\,(p_B + p_{D^*})_\rho\, q_\sigma\,, \tag{23b}$$

$$\big\langle D^* \big|\, \bar{c}\gamma^\mu\gamma^5 b \,\big|\overline{B}\big\rangle \equiv \varepsilon^{*\mu} f + a_+\,\varepsilon^* \cdot p_B\,(p_B + p_{D^*})^\mu + a_-\,\varepsilon^* \cdot p_B\, q^\mu\,, \tag{23c}$$

$$\begin{aligned}\big\langle D^* \big|\, \bar{c}\sigma^{\mu\nu} b \,\big|\overline{B}\big\rangle \equiv{}& -a_{T_+}\,\epsilon^{\mu\nu\rho\sigma}\varepsilon^*_\rho (p_B + p_{D^*})_\sigma - a_{T_-}\,\epsilon^{\mu\nu\rho\sigma}\varepsilon^*_\rho\, q_\sigma \\ & - a_{T_0}\,\varepsilon^* \cdot p_B\,\epsilon^{\mu\nu\rho\sigma}(p_B + p_{D^*})_\rho\, q_\sigma\,.\end{aligned} \tag{23d}$$

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. [19], via

$$a_0 = -\sqrt{r_{D^*}} h_P \,, \tag{24a}$$
$$f = \sqrt{r_{D^*}}(w+1) m_B h_{A_1} \,, \tag{24b}$$
$$g = \frac{h_{\rm V}}{2\sqrt{r_{D^*}} m_B} \,, \tag{24c}$$
$$a_- = \frac{h_{A_3} - r_{D^*} h_{A_2}}{2\sqrt{r_{D^*}} m_B} \,, \tag{24d}$$
$$a_+ = -\frac{r_{D^*} h_{A_2} + h_{A_3}}{2\sqrt{r_{D^*}} m_B} \,, \tag{24e}$$
$$a_{T_0} = \frac{h_{T_3}}{2\sqrt{r_{D^*}} m_B^2} \,, \tag{24f}$$
$$a_{T_-} = \frac{(1-r_{D^*}) h_{T_1} - (r_{D^*}+1) h_{T_2}}{2\sqrt{r_{D^*}}} \,, \tag{24g}$$
$$a_{T_+} = \frac{(1-r_{D^*}) h_{T_2} - (r_{D^*}+1) h_{T_1}}{2\sqrt{r_{D^*}}} \,. \tag{24h}$$

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. (24).

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

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

$$\mathrm{FF}_{D_0^*} = \Big\{ g_P,\, g_+,\, g_-,\, g_T \Big\} \,, \tag{25a}$$
$$\mathrm{FF}_{D_1^*} = \Big\{ g_S,\, g_{V_1},\, g_{V_2},\, g_{V_3},\, g_a,\, g_{T_1},\, g_{T_2},\, g_{T_3} \Big\} \,, \tag{25b}$$
$$\mathrm{FF}_{D_1} = \Big\{ f_S,\, f_{V_1},\, f_{V_2},\, f_{V_3},\, f_a,\, f_{T_1},\, f_{T_2},\, f_{T_3} \Big\} \,, \tag{25c}$$
$$\mathrm{FF}_{D_2^*} = \Big\{ k_P,\, k_{A_1},\, k_{A_2},\, k_{A_3},\, k_V,\, k_{T_1},\, k_{T_2},\, k_{T_3} \Big\} \,, \tag{25d}$$

which following Ref. [23], are defined for $\overline{B} \to D_0^*$ via

$$\begin{aligned}
\big\langle D_0^* \big|\, \bar{c}\, b \,\big|\overline{B}\big\rangle &= \big\langle D_0^* \big|\, \bar{c}\gamma_\mu b \,\big| B\big\rangle = 0 \,, \\
\big\langle D_0^* \big|\, \bar{c}\gamma_5 b \,\big|\overline{B}\big\rangle &= \sqrt{m_{D_0^*} m_B}\, g_P \,, \\
\big\langle D_0^* \big|\, \bar{c}\gamma_\mu\gamma_5 b \,\big|\overline{B}\big\rangle &= \sqrt{m_{D_0^*} m_B}\big[g_+(v_\mu + v'_\mu) + g_-(v_\mu - v'_\mu) \big], \\
\big\langle D_0^* \big|\, \bar{c}\sigma_{\mu\nu} b \,\big|\overline{B}\big\rangle &= \sqrt{m_{D_0^*} m_B}\, g_T\, \varepsilon_{\mu\nu\alpha\beta}\, v^\alpha v'^\beta \,,
\end{aligned} \tag{26a}$$

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

$$\big\langle D_1^* \big|\, \bar{c}\, b \,\big|\overline{B}\big\rangle = -\sqrt{m_{D_1^*} m_B}\, g_S\, (\epsilon^* \cdot v) \,,$$

$$
\begin{aligned}
\langle D_1^*|\,\bar{c}\gamma_5 b\,|\overline{B}\rangle &= 0\,,\\
\langle D_1^*|\,\bar{c}\gamma_\mu b\,|\overline{B}\rangle &= \sqrt{m_{D_1^*} m_B}\,\left[g_{V_1}\,\epsilon_\mu^* + (g_{V_2} v_\mu + g_{V_3} v'_\mu)\,(\epsilon^*\cdot v)\right],\\
\langle D_1^*|\,\bar{c}\gamma_\mu\gamma_5 b\,|\overline{B}\rangle &= i\sqrt{m_{D_1^*} m_B}\,g_A\,\varepsilon_{\mu\alpha\beta\gamma}\,\epsilon^{*\alpha} v^\beta\, v'^\gamma\,,\\
\langle D_1^*|\,\bar{c}\sigma_{\mu\nu} b\,|\overline{B}\rangle &= i\sqrt{m_{D_1^*} m_B}\,\left[g_{T_1}(\epsilon_\mu^* v_\nu - \epsilon_\nu^* v_\mu) + g_{T_2}(\epsilon_\mu^* v'_\nu - \epsilon_\nu^* v'_\mu) + g_{T_3}(\epsilon^*\cdot v)(v_\mu v'_\nu - v_\nu v'_\mu)\right].
\end{aligned}
\tag{26b}
$$

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

$$
\begin{aligned}
\langle D_1|\,\bar{c}\,b\,|\overline{B}\rangle &= \sqrt{m_{D_1} m_B}\,f_S\,(\epsilon^*\cdot v)\,,\\
\langle D_1|\,\bar{c}\gamma_5 b\,|\overline{B}\rangle &= 0\,,\\
\langle D_1|\,\bar{c}\gamma_\mu b\,|\overline{B}\rangle &= \sqrt{m_{D_1} m_B}\,\left[f_{V_1}\,\epsilon_\mu^* + (f_{V_2} v_\mu + f_{V_3} v'_\mu)(\epsilon^*\cdot v)\right],\\
\langle D_1|\,\bar{c}\gamma_\mu\gamma_5 b\,|\overline{B}\rangle &= i\,\sqrt{m_{D_1} m_B}\,f_A\,\varepsilon_{\mu\alpha\beta\gamma}\epsilon^{*\alpha} v^\beta v'^\gamma\,,\\
\langle D_1|\,\bar{c}\sigma_{\mu\nu} b\,|B\rangle &= i\sqrt{m_{D_1} m_B}\big[f_{T_1}(\epsilon_\mu^* v_\nu - \epsilon_\nu^* v_\mu) + f_{T_2}(\epsilon_\mu^* v'_\nu - \epsilon_\nu^* v'_\mu) + f_{T_3}(\epsilon^*\cdot v)(v_\mu v'_\nu - v_\nu v'_\mu)\big],
\end{aligned}
\tag{26c}
$$

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

$$
\begin{aligned}
\langle D_2^*|\,\bar{c}\,b\,|\overline{B}\rangle &= 0\,,\\
\langle D_2^*|\,\bar{c}\gamma_5 b\,|\overline{B}\rangle &= \sqrt{m_{D_2^*} m_B}\,k_P\,\epsilon_{\alpha\beta}^*\,v^\alpha v^\beta\,,\\
\langle D_2^*|\,\bar{c}\gamma_\mu b\,|\overline{B}\rangle &= i\sqrt{m_{D_2^*} m_B}\,k_V\,\varepsilon_{\mu\alpha\beta\gamma}\,\epsilon^{*\alpha\sigma} v_\sigma v^\beta v'^\gamma\,,\\
\langle D_2^*|\,\bar{c}\gamma_\mu\gamma_5 b\,|\overline{B}\rangle &= \sqrt{m_{D_2^*} m_B}\,\left[k_{A_1}\,\epsilon_{\mu\alpha}^* v^\alpha + (k_{A_2} v_\mu + k_{A_3} v'_\mu)\,\epsilon_{\alpha\beta}^*\,v^\alpha v^\beta\right],\\
\langle D_2^*|\,\bar{c}\sigma_{\mu\nu} b\,|\overline{B}\rangle &= \sqrt{m_{D_2^*} m_B}\,\varepsilon_{\mu\nu\alpha\beta}\left\{[k_{T_1}(v+v')^\alpha + k_{T_2}(v-v')^\alpha)]\,\epsilon^{*\gamma\beta} v_\gamma + k_{T_3}\,v^\alpha v'^\beta \epsilon^{*\rho\sigma} v_\rho v_\sigma\right\}.
\end{aligned}
\tag{26d}
$$

(NB: In the case of the `ISGW2` FF parametrization for the D_1 and D_1^*, `EvtGen` includes an additional *ad hoc* 'smearing' by the factor $\sqrt{q^2_{\rm max,mean}/q^2_{\rm max}}$ on each form factor. This is included by default, but can be deactivated via the bool setting `"SmearQ2"`.)

4. $\Lambda_b\to\Lambda_c$

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

$$
{\rm FF}_{\Lambda_c} = \Big\{h_S,\,h_P,\,f_1,\,f_2,\,f_3,\,g_1,\,g_2,\,g_3,\,h_1,\,h_2,\,h_3,\,h_4\Big\}\,,
\tag{27}
$$

The form factors are defined as in Ref. [27], using the sign convention ${\rm Tr}[\gamma^\mu\gamma^\nu\gamma^\sigma\gamma^\rho\gamma^5] = -4i\epsilon^{\mu\nu\rho\sigma}$, via

$$
\begin{aligned}
\langle\Lambda_c(p',s')|\bar{c}\,b|\Lambda_b(p,s)\rangle &= h_S\,\bar{u}(p',s')\,u(p,s)\,, &(28a)\\
\langle\Lambda_c(p',s')|\bar{c}\gamma_5 b|\Lambda_b(p,s)\rangle &= h_P\,\bar{u}(p',s')\,\gamma_5\,u(p,s)\,, &(28b)\\
\langle\Lambda_c(p',s')|\bar{c}\gamma_\nu b|\Lambda_b(p,s)\rangle &= \bar{u}(p',s')\big[f_1\gamma_\mu + f_2 v_\mu + f_3 v'_\mu\big]u(p,s)\,, &(28c)\\
\langle\Lambda_c(p',s')|\bar{c}\gamma_\nu\gamma_5 b|\Lambda_b(p,s)\rangle &= \bar{u}(p',s')\big[g_1\gamma_\mu + g_2 v_\mu + g_3 v'_\mu\big]\gamma_5\,u(p,s)\,, &(28d)\\
\langle\Lambda_c(p',s')|\bar{c}\,\sigma_{\mu\nu}\,b|\Lambda_b(p,s)\rangle &= \bar{u}(p',s')\big[h_1\,\sigma_{\mu\nu} + i\,h_2(v_\mu\gamma_\nu - v_\nu\gamma_\mu) + i\,h_3(v'_\mu\gamma_\nu - v'_\nu\gamma_\mu) &(28e)
\end{aligned}
$$

$$+\, i\, h_4(v_\mu v'_\nu - v_\nu v'_\mu)\big] u(p,s)\,. \tag{28f}$$

The spinors are normalized to $\bar{u}(p,s)u(p,s) = 2m$. (Note that another common definition for the SM form factors is [41]

$$\begin{aligned}\langle \Lambda_c(p',s')|\bar{c}\gamma_\mu b|\Lambda_b(p,s)\rangle &= \bar{u}(p',s')\big[F_1\,\gamma_\mu - iF_2\,\sigma_{\mu\nu}\,q^\nu + F_3\,q_\mu\big]u(p,s)\,,\\ \langle \Lambda_c(p',s')|\bar{c}\gamma_\mu\gamma_5 b|\Lambda_b(p,s)\rangle &= \bar{u}(p',s')\big[G_1\,\gamma_\mu - iG_2\,\sigma_{\mu\nu}\,q^\nu + G_3\,q_\mu\big]\gamma_5\,u(p,s)\,.\end{aligned} \tag{29}$$

The notation of Ref. [41] also exchanges upper and lowercase symbols – i.e. $F_i \leftrightarrow f_i$ and $G_i \leftrightarrow g_i$ – with respect to Eqs. (28) and (29).)

5. $B \to \rho,\, \omega$

The $B \to \rho$ or ω decays have form factor tensor with ordered components

$$\mathrm{FF}_{\rho,\omega} = \Big\{A_P, V, A_0, A_1, A_{12}, T_1, T_2, T_{23}\Big\}\,, \tag{30}$$

which are defined as in Ref. [25]. Explicitly,

$$\sqrt{2}\big\langle V\big|\,\bar{u}\gamma^5 b\,\big|\overline{B}\big\rangle \equiv A_P\,\varepsilon^*\cdot q\ , \tag{31a}$$

$$\sqrt{2}\big\langle V\big|\,\bar{u}\gamma^\mu b\,\big|\overline{B}\big\rangle \equiv -\frac{iV}{m_B+m_V}\,\epsilon^{\mu\nu\rho\sigma}\,\varepsilon^*_\nu\,(p_B+p_V)_\rho\,q_\sigma\,, \tag{31b}$$

$$\sqrt{2}\big\langle V\big|\,\bar{u}\gamma^\mu\gamma^5 b\,\big|\overline{B}\big\rangle \equiv A_1(m_B+m_V)\varepsilon^{*\mu} - A_2\frac{(p_B+p_V)^\mu\,\varepsilon^*\cdot q}{m_B+m_V} \tag{31c}$$

$$+\,\frac{\varepsilon^*\cdot q\,q^\mu}{q^2}\Big[A_2(m_B-m_V) - A_1(m_B+m_V) + 2m_V A_0\Big]\,, \tag{31d}$$

$$\sqrt{2}\big\langle V\big|\,\bar{u}\sigma^{\mu\nu} b\,\big|\overline{B}\big\rangle \equiv \epsilon^{\mu\nu\rho\sigma}\Big[T_1\varepsilon^*_\rho(p_B+p_V)_\sigma - (T_2+T_1)\frac{m_B^2-m_V^2}{q^2}\varepsilon^*_\rho q_\sigma \tag{31e}$$

$$+\,(p_B+p_V)_\rho q_\sigma\frac{\varepsilon^*\cdot q}{q^2}\left((T_1+T_2) + T_3\frac{q^2}{m_B^2-m_V^2}\right)\Big]\,, \tag{31f}$$

with the additional redefinitions with respect to A_{12} and T_{23},

$$A_2 = \frac{A_1(m_B^2-m_V^2-q^2)(m_B+m_V)^2 - 16A_{12}m_B m_V^2(m_B+m_V)}{4|p_V|^2 m_B^2}\,, \tag{32a}$$

$$T_3 = \frac{T_2(m_B^2+3m_V^2-q^2)(m_B^2-m_V^2) - 8T_{23}m_B m_V^2(m_B-m_V)}{4|p_V|^2 m_B^2}\,, \tag{32b}$$

with $|p_V|$ the vector meson 3-momentum in the B rest frame.

Light-cone sum rule results (LCSR) results are available for both the SM and NP form factors, parametrized by an optimized z expansion, in the form of a correlated, beyond zero recoil fit between the SM and NP form factors [25]. This LCSR-based parametrization is referred to as '`BSZ`'.

6. $\Lambda_b \to \Lambda_c^*(2595)$ *and* $\Lambda_c^*(2625)$

The $\Lambda_b^0 \to \Lambda_c^*$ form factor tensors have ordered components

$$\mathrm{FF}_{\Lambda_c^*(2595)} = \Big\{ d_S, d_P, d_{V1}, d_{V2}, d_{V3}, d_{A1}, d_{A2}, d_{A3}, d_{T1}, d_{T2}, d_{T3}, d_{T4} \Big\} , \tag{33a}$$

$$\mathrm{FF}_{\Lambda_c^*(2625)} = \Big\{ l_S, l_P, l_{V1}, l_{V2}, l_{V3}, l_{V4}, l_{A1}, l_{A2}, l_{A3}, l_{A4}, l_{T1}, l_{T2}, l_{T3}, l_{T4}, l_{T6}, l_{T7} \Big\} . \tag{33b}$$

following the conventions and definitions in Ref. [30]. The form factor l_{T5} must be eliminated, according to the kernel of the $\Lambda_b \to \Lambda_c^*(2625)$ amplitudes, after matching onto HQET or a particular model of interest. See Ref. [30].

Explicitly, representing Λ_b and $\Lambda_c^*(2595)$ by spinors $u_b(p,s)$ and $\bar{u}_c(p',s')$, respectively, with momenta $p = m_{\Lambda_b} v$ and $p' = m_{\Lambda_c^*} v'$, the form factors d_X are defined via

$$\begin{aligned}
\langle \Lambda_c^*(2595)|\bar{c}\, b|\Lambda_b\rangle &= -d_S\, \bar{u}_c \gamma_5 u_b \,, \\
\langle \Lambda_c^*(2595)|\bar{c}\gamma_5 b|\Lambda_b\rangle &= -d_P\, \bar{u}_c\, u_b \,, \\
\langle \Lambda_c^*(2595)|\bar{c}\gamma_\mu b|\Lambda_b\rangle &= \bar{u}_c \big[d_{V1}\gamma_\mu + d_{V2} v_\mu + d_{V3} v'_\mu \big] \gamma_5 u_b \,, \\
\langle \Lambda_c^*(2595)|\bar{c}\gamma_\mu\gamma_5 b|\Lambda_b\rangle &= \bar{u}_c \big[d_{A1}\gamma_\mu + d_{A2} v_\mu + d_{A3} v'_\mu \big] u_b \,, \\
\langle \Lambda_c^*(2595)|\bar{c}\, \sigma_{\mu\nu}\, b|\Lambda_b\rangle &= -\bar{u}_c \big[d_{T1}\, \sigma_{\mu\nu} + i\, d_{T2} v_{[\mu}\gamma_{\nu]} + i\, d_{T3} v'_{[\mu}\gamma_{\nu]} + i\, d_{T4} v_{[\mu} v'_{\nu]} \big] \gamma_5 u_b \,,
\end{aligned} \tag{34}$$

The charmed spin-3/2 state is represented by a Rarita-Schwinger tensor, $\Psi_c^\mu(p',s')$, satisfying the usual transversity and projective conditions $v'\cdot\Psi_c = 0$ and $\gamma\cdot\Psi_c = 0$. The form factors l_X are then defined via

$$\begin{aligned}
\langle \Lambda_c^*(2625)|\bar{c}\, b|\Lambda_b\rangle &= l_S\, v\cdot\overline{\Psi}_c u_b \,, \\
\langle \Lambda_c^*(2625)|\bar{c}\gamma_5 b|\Lambda_b\rangle &= l_P\, v\cdot\overline{\Psi}_c \gamma_5 u_b \,, \\
\langle \Lambda_c^*(2625)|\bar{c}\gamma_\mu b|\Lambda_b\rangle &= v\cdot\overline{\Psi}_c \big[l_{V1}\gamma_\mu + l_{V2} v_\mu + l_{V3} v'_\mu \big] u_b + l_{V4} \overline{\Psi}_{c\mu} u_b \,, \\
\langle \Lambda_c^*(2625)|\bar{c}\gamma_\mu\gamma_5 b|\Lambda_b\rangle &= v\cdot\overline{\Psi}_c \big[l_{A1}\gamma_\mu + l_{A2} v_\mu + l_{A3} v'_\mu \big] \gamma_5 u_b + l_{A4} \overline{\Psi}_{c\mu} \gamma_5 u_b \,, \\
\langle \Lambda_c^*(2625)|\bar{c}\, \sigma_{\mu\nu}\, b|\Lambda_b\rangle &= v\cdot\overline{\Psi}_c \big[l_{T1}\, \sigma_{\mu\nu} + i\, l_{T2} v_{[\mu}\gamma_{\nu]} + i\, l_{T3} v'_{[\mu}\gamma_{\nu]} + i\, l_{T7} v_{[\mu} v'_{\nu]} \big] u_b \\
&\quad + i\, \overline{\Psi}_{c[\mu} \big[l_{T4}\, \gamma_{\nu]} + l_{T5} v_{\nu]} + l_{T6} v'_{\nu]} \big] u_b \,.
\end{aligned} \tag{35}$$

E. Form Factor uncertainties

At present, e.g. the `BGLVar` parametrization permits via `setFFEigenvectors` functionality (see Sec. III J) 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). That is, the covariance matrix is set to the identity, and the basis of variations

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

Similary the $a_i^{f_+}$ and $a_i^{f_0}$, $i = 0, \ldots, 3$ parameters are directly manipulated 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"} .
```

By contrast, the linearized form of the $B \to \rho$, ω LCSR-based parametrization, referred to as '`BSZVar`', features a non-trivial covariance matrix. At present we include just the first eight principal directions of the 21 parameter fit, in the basis

```
{"delta_e1","delta_e2","delta_e3","delta_e4","delta_e5","delta_e6",
      "delta_e7","delta_e8"} .
```

Each covariance eigenvector e_i is normalized by the square-root of its eigenvalue $\sqrt{\lambda_i}$, so that unit variation in each `"delta_ei"` corresponds to a 1σ variation.

A large number of other `Var` classes are available in the library: the implemented basis of parameters may be easily read off each class implementation.

F. D^{**} strong decays

The library incorporates the strong decays $D_1 \to D^*\pi$, $D_2^* \to D^{(*)}\pi$. While the latter can proceed only via d-wave, the former proceeds by d-wave at leading order in HQET but may include s-wave contributions at subleading order [42–44], that may be thought of a contribution arising from D_1^*–D_1 mixing. When D^{**} decays are included in a run, a 'form-factor' parametrization for them must be specified in each FF scheme: At present the only partial-wave parametrization '`PW`' is included.

The explicit $D_1 \to D^*\pi$ partial-wave amplitude

$$\mathcal{A}^{\lambda\kappa}_{D_1^+ \to D^*\pi} = \frac{1}{f_\pi}\left(\frac{D}{\sqrt{6}} + \sqrt{\frac{3}{2}}S\right)\left[\frac{E^*_{D^*} - m_{D^*}}{m_{D_1}}\varepsilon_1^{-\lambda}\cdot p_\pi \; \varepsilon_*^{\kappa}\cdot p_\pi + |\boldsymbol{p}_\pi|^2 \varepsilon_1^{-\lambda}\cdot\varepsilon_*^{\kappa}\right] \tag{36}$$

$$+ \frac{D}{\sqrt{6}}\frac{3m_{D^*}}{m_{D_1}}\varepsilon_1^{-\lambda}\cdot p_\pi \; \varepsilon_*^{\kappa}\cdot p_\pi \,, \tag{37}$$

in which $|\boldsymbol{p}_\pi|$ is the pion momentum in the D_1 frame, and the S and $D \in \mathbb{C}$ parametrize the s- and d-wave contributions, respectively. This is equivalent to the conventions in Ref. [44], but with S scaled by an additional factor $-3/\sqrt{2}$ with respect to D to match the conventions of the `EvtGen` '`VVSPWave`' class. (S is normalized by an additional $|\boldsymbol{p}_\pi|$ factor so that it is dimensionless.) The corresponding partial rate $\Gamma_{D_1\to D^*\pi} = (|D|^2 + 9|S|^2/2)|\boldsymbol{p}_\pi|^5/(24\pi f_\pi^2 m_{D_1}^2)$. The parameters S and D are treated as constant form-factors by the library, and may be set as options of the '`PW`' parametrization. This permits reweighting between specific d/s-wave admixtures. The default is $S = 0$, $D = 1$.

The explicit $D_2^* \to D^{(*)}\pi$ partial-wave amplitudes, in the same notation are

$$\mathcal{A}^{\lambda\kappa}_{D_2^{*+} \to D^*\pi} = \frac{D}{f_\pi} i\epsilon^{\alpha\beta\gamma\delta}\varepsilon^{-\lambda}_{\alpha\tau} p_\pi^\tau \epsilon^{\kappa}_{D^*\beta} p_{\pi\gamma} p_{D^*\delta} \,, \tag{38}$$

$$\mathcal{A}^{\lambda}_{D_2^{*+}\to D\pi} = \frac{D}{f_\pi}\varepsilon^{-\lambda}_{\mu\nu}p^{\mu}_{\pi}p^{\nu}_{\pi}\,, \tag{39}$$

$$\tag{40}$$

corresponding to the partial rates $\Gamma_{D_2^*\to D^*\pi} = |D|^2|\boldsymbol{p}_\pi|^5/(40\pi f_\pi^2 m^2_{D_1})$ and $\Gamma_{D_2^*\to D\pi} = |D|^2|\boldsymbol{p}_\pi|^5/(60\pi f_\pi^2 m^2_{D_1})$, respectively. The parameter D is treated as a constant form-factor by the library, and may be set as an option of the '`PW`' parametrization. The default is $D = 1$.

G. Resonance Lineshapes

`EvtGen` includes additional momentum and angular momentum dependent models for resonance lineshapes, that can be numerically non-negligible for broad resonances such as the D^{**}. These (somewhat arbitrary) lineshape models typically factorize from the decay amplitudes themselves, and are generically invariant under reweighting. They are therefore not presently included automatically the `Hammer` library: Effects of reweighting on the lineshape model, if important, can instead be included by the user via `setEventBaseWeight`. The latter may be required in two cases:

- `EvtGen` does not incorporate the lineshape in the case the decay is pure phase space. Thus caution should be used when reweighting broad resonances from pure phase generated by `EvtGen`, because lineshape weights typically included by `EvtGen` will be absent.
- The `EvtGen` lineshape models are sensitive to the angular momentum of the resonance. Thus reweighting that alters an admixture of partial waves (such as is possible with the `PW` FF parametrization of the $D_1 \to D^*\pi$ decay) may incorporate a different lineshape than would have been generated directly by `EvtGen`.

For more information on lineshape options in `EvtGen` we refer to its documentation directly [45].

H. τ spinors

I. $D^{(*,**)}$ polarizations, $\Lambda_c^{(*)}$ 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.
- `-DMAX_CXX_STANDARD=[14,17,20]`: select the maximum C++ dialect used by the compiler. The configuration step determines the dialect by taking the minimum among the maximum supported by the compiler, the value of this option and the dialect used in compiling ROOT if `WITH_ROOT` is enabled. The minimum allowed dialect is always C++14.

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] P. Krawczyk and S. Pokorski, Phys. Rev. Lett. **60**, 182 (1988).

[2] P. Heiliger and L. M. Sehgal, Phys. Lett. **B229**, 409 (1989).

[3] J. Kalinowski, Phys. Lett. **B245**, 201 (1990).

[4] B. Grzadkowski and W.-S. Hou, Phys. Lett. **B272**, 383 (1991).

[5] Y. Grossman and Z. Ligeti, Phys. Lett. **B332**, 373 (1994), arXiv:hep-ph/9403376 [hep-ph].

[6] M. Tanaka, Z. Phys. **C67**, 321 (1995), arXiv:hep-ph/9411405 [hep-ph].

[7] W. D. Goldberger, (1999), arXiv:hep-ph/9902311 [hep-ph].

[8] D. Buskulic *et al.* (ALEPH Collaboration), Phys. Lett. **B298**, 479 (1993).
[9] Y. S. Amhis *et al.* (HFLAV), (2019), arXiv:1909.12524 [hep-ex].
[10] J. P. Lees *et al.* (BaBar Collaboration), Phys. Rev. **D88**, 072012 (2013), arXiv:1303.0571 [hep-ex].
[11] R. Aaij *et al.* (LHCb Collaboration), Phys. Rev. Lett. **115**, 111803 (2015), [Addendum: Phys. Rev. Lett. 115, no.15, 159901 (2015)], arXiv:1506.08614 [hep-ex].
[12] M. Huschle *et al.* (Belle Collaboration), Phys. Rev. **D92**, 072014 (2015), arXiv:1507.03233 [hep-ex].
[13] B. Grinstein and A. Kobach, Phys. Lett. **B771**, 359 (2017), arXiv:1703.08170 [hep-ph].
[14] C. G. Boyd, B. Grinstein, and R. F. Lebed, Nucl. Phys. **B461**, 493 (1996), arXiv:hep-ph/9508211 [hep-ph].
[15] C. G. Boyd, B. Grinstein, and R. F. Lebed, Phys. Rev. **D56**, 6895 (1997), arXiv:hep-ph/9705252 [hep-ph].
[16] D. Scora and N. Isgur, Phys. Rev. **D52**, 2783 (1995), arXiv:hep-ph/9503486 [hep-ph].
[17] N. Isgur, D. Scora, B. Grinstein, and M. B. Wise, Phys. Rev. **D39**, 799 (1989).
[18] I. Caprini, L. Lellouch, and M. Neubert, Nucl. Phys. **B530**, 153 (1998), arXiv:hep-ph/9712417 [hep-ph].
[19] F. U. Bernlochner, Z. Ligeti, M. Papucci, and D. J. Robinson, Phys. Rev. **D95**, 115008 (2017), arXiv:1703.05330 [hep-ph].
[20] F. U. Bernlochner, Z. Ligeti, M. Papucci, M. T. Prim, D. J. Robinson, and C. Xiong, (2022), arXiv:2206.11281 [hep-ph].
[21] A. K. Leibovich, Z. Ligeti, I. W. Stewart, and M. B. Wise, Phys. Rev. **D57**, 308 (1998), arXiv:hep-ph/9705467 [hep-ph].
[22] A. K. Leibovich, Z. Ligeti, I. W. Stewart, and M. B. Wise, Phys. Rev. Lett. **78**, 3995 (1997), arXiv:hep-ph/9703213 [hep-ph].
[23] F. U. Bernlochner, Z. Ligeti, and D. J. Robinson, (2017), arXiv:1711.03110 [hep-ph].
[24] F. U. Bernlochner and Z. Ligeti, Phys. Rev. **D95**, 014022 (2017), arXiv:1606.09300 [hep-ph].
[25] A. Bharucha, D. M. Straub, and R. Zwicky, JHEP **08**, 098 (2016), arXiv:1503.05534 [hep-ph].
[26] M. Pervin, W. Roberts, and S. Capstick, Phys. Rev. **C72**, 035201 (2005), arXiv:nucl-th/0503030 [nucl-th].
[27] F. U. Bernlochner, Z. Ligeti, D. J. Robinson, and W. L. Sutcliffe, Phys. Rev. **D99**, 055008 (2019), arXiv:1812.07593 [hep-ph].
[28] F. U. Bernlochner, Z. Ligeti, D. J. Robinson, and W. L. Sutcliffe, Phys. Rev. Lett. **121**, 202001 (2018), arXiv:1808.09464 [hep-ph].
[29] A. K. Leibovich and I. W. Stewart, Phys. Rev. **D57**, 5620 (1998), arXiv:hep-ph/9711257 [hep-ph].
[30] M. Papucci and D. J. Robinson, (2021), arXiv:2105.09330 [hep-ph].
[31] V. Kiselev, (2002), arXiv:hep-ph/0211021.

[32] D. Ebert, R. Faustov, and V. Galkin, Phys. Rev. D **68**, 094020 (2003), arXiv:hep-ph/0306306.
[33] T. D. Cohen, H. Lamm, and R. F. Lebed, Phys. Rev. D **100**, 094503 (2019), arXiv:1909.10691 [hep-ph].
[34] J. A. Bailey *et al.* (Fermilab Lattice, MILC), Phys. Rev. D **92**, 014024 (2015), arXiv:1503.07839 [hep-lat].
[35] N. Gubernari, A. Kokulu, and D. van Dyk, JHEP **01**, 150 (2019), arXiv:1811.00983 [hep-ph].
[36] J. H. Kuhn and E. Mirkes, Z. Phys. **C56**, 661 (1992), [Erratum: Z. Phys.C67,364(1995)].
[37] O. Shekhovtsova, T. Przedzinski, P. Roig, and Z. Was, Phys. Rev. **D86**, 113008 (2012), arXiv:1203.3955 [hep-ph].
[38] I. M. Nugent, T. Przedzinski, P. Roig, O. Shekhovtsova, and Z. Was, Phys. Rev. **D88**, 093012 (2013), arXiv:1310.1053 [hep-ph].
[39] E. Barberio, B. van Eijk, and Z. Wąs, Computer Physics Communications **66**, 115 (1991).
[40] Z. Ligeti, M. Papucci, and D. J. Robinson, JHEP **01**, 083 (2017), arXiv:1610.02045 [hep-ph].
[41] A. F. Falk and M. Neubert, Phys. Rev. **D47**, 2982 (1993), arXiv:hep-ph/9209269 [hep-ph].
[42] R. Casalbuoni, A. Deandrea, N. Di Bartolomeo, R. Gatto, F. Feruglio, and G. Nardulli, Phys. Rept. **281**, 145 (1997), arXiv:hep-ph/9605342.
[43] U. Kilian, J. G. Korner, and D. Pirjol, Phys. Lett. B **288**, 360 (1992).
[44] M. Lu, M. B. Wise, and N. Isgur, Phys. Rev. D **45**, 1553 (1992).
[45] A. Ryd, D. Lange, N. Kuznetsova, S. Versille, M. Rotondo, D. P. Kirkby, F. K. Wuerthwein, and A. Ishikawa, (2005).