Eur. Phys. J. C (2025) 85:1448
https://doi.org/10.1140/epjc/s10052-025-15108-7

THE EUROPEAN PHYSICAL JOURNAL C

Regular Article - Computing, Software and Data Science

Rapid event extraction and tensorial event adaption

Libraries for efficient access and generic reweighting of parton-level events and their implementation in the MADTREX module

Stefan Roiser[1,a], **Robert Schöfbeck**[2,b], **Zenny Wettersten**[1,2,c]

[1] CERN, Esplanade des Particules 1, Geneva 1211, Switzerland
[2] MBI (HEPHY), ÖAW, Dominikanerbastei 16, Vienna 1010, Austria

Received: 10 October 2025 / Accepted: 21 November 2025

Abstract We present `Rex` and `teaRex`, C++17 libraries for efficient management of parton-level hard scattering event information and completely generic reweighting of such events, respectively. `Rex` is primarily an interfacing and I/O library for Les Houches Event format files and provides an internal event format designed with data parallelism in mind, and `teaRex` extends this format to provide full parton-level reweighting functionality with minimal code needing to be written by the end user. These libraries serve as the foundation for the MADTREX reweighting module for MADGRAPH5_AMC@NLO, extending the functionality of the `CUDACPP` plugin to allow for data-parallel model-generic leading order parameter reweighting with on-CPU hardware acceleration and GPU offloading, speeding up reweighting by more than two orders of magnitude compared to MADGRAPH5_AMC@NLO running on the exact same hardware while providing trivial scalability to larger and distributed systems.

1 Introduction

Adoption of explicit data-parallel hardware acceleration for high-energy physics (HEP) software — using on-CPU *single instruction across multiple data streams* (SIMD) instruction sets and off-CPU *single instruction across multiple threads* (SIMT) GPU offloading[1] — has in recent years not only proven to be of great importance in the face of impending computational needs, but also a very difficult task [1–4]. Significant work has been put into porting existing or writing new HEP software to properly utilise existing and upcoming hardware [5–17]. However, these efforts generally target specific codebases and implementations, leaving common issues — such as the interfacing between older, typically object-oriented (OO) data formats used in existing codebases and data-oriented structures-of-arrays (SoA), more fit for data parallelism — a problem re-implemented across different collaborations.

Simultaneously, improved experimental precision results in an increase in the size of necessary theoretical event samples for statistical comparisons with the standard model (SM). The efficient reuse of existing samples is thus ever more important. In SM studies this can occur e.g. when estimating simulation uncertainties by evaluating different parton distribution function (pdf) sets for the initial-state particles in a parton-level hard scattering event and varying the factorisation and renormalisation scales they are evaluated at; alternatively, when studying the phenomenology of beyond-the-SM (BSM) models with their infinite available parameter spaces it is unfeasible to run the full simulation chain for samples at all parameters of all models, and instead the

[1] We forego details on SIMD and SIMT parallelism as well as the differences between object-oriented arrays of structures and data-oriented structures of arrays here, but a quick internet search will provide extensive explanations. As a short description, arrays of structures can be thought of as objects used in chronologically ordered for-loops, while SoAs are flipped such that equivalent data are adjacent in memory to allow the same for-loop to be ordered in space rather than time; with this in mind, SIMD and SIMT architectures allow for this exact type of data parallelism where the same compute instruction is performed across many equivalent data at the same time. See also footnote 3.

[a] e-mail: stefan.roiser@cern.ch
[b] e-mail: robert.schoefbeck@oeaw.ac.at
[c] e-mail: zenny.wettersten@cern.ch (corresponding author)

Published online: 21 December 2025

Monte Carlo event weights of an existing sample can be re-evaluated for the new model and propagated to the end experimental observables in a procedure known as matrix element reweighting (henceforth called parameter reweighting).

We attempt to address both of these issues with two new C++17 libraries: `Rex`, the `R`apid `E`vent e`X`traction library; and `teaRex`, a library for `T`ensorial `E`vent `A`daption with `Rex`. The former is intended to provide interfacing between OO data formats following the conventions of the Les Houches Event (LHE) file format [18–20] and SoA formats with event data stored in contiguous vectors; in the `Rex` format, the latter are split into independent SoAs for individual subprocesses based on user-provided event categorisation. Version 1.0.0 `Rex` comes shipped with I/O routines for the XML-based LHE standard, but provides interfaces for the implementation of custom readers and writers for generic file formats with little user-side effort. On the other hand, `teaRex` is a minimal extension to `Rex` providing a structure for generic parton-level event reweighting with minimal interfacing necessary from developers. `teaRex` uses the event sorting capabilities and SoA event structure from `Rex` to enable immediate SIMD- and SIMT-friendly data access to event data, which is used to automate the full parton-level event reweighting process in a completely generic way. This can be done for any user-supplied reweighting function, whether it be for parameter, pdf, or any other type of reweighting. Additionally, `Rex` and `teaRex` are used as a foundation for the MadtRex module, which repurposes the data-parallel scattering amplitudes generated by the `CUDACPP` plugin [12–17] for MadGraph5_aMC@NLO (MG5aMC) [21] to create executables for generic tree-level parameter reweighting, overriding the original MG5aMC reweighting module [22]. Using on-CPU SIMD instructions and GPU offloading, MadtRex increases peak reweighting throughput for computationally heavy processes by more than two orders of magnitude when compared to MG5aMC, but even without any explicitly implemented data parallelism on-CPU MadtRex executables without SIMD instructions increase event throughput by roughly a factor 30-60. This "automatic" speed-up can be attributed to a combination of several factors differentiating MadtRex from the upstream MG5aMC reweighting module: the better-scaling sorting algorithm used by `Rex` and `teaRex`; running a singular compiled executable rather than through an interpreted language; and compiler optimisations including but not limited to automatically applied multithreading.

This paper is split into three main sections: section 2 provides a detailed description of the `Rex` library, a usage manual for applying it to other C++ programs, and benchmarks for the included LHE file format reader and sorting algorithm; `teaRex` is then presented in section 3, also with a usage manual as well as a sketch for how to apply it specifically for pdf reweighting; and finally a usage guide for MadtRex is given in section 4 with throughput comparisons to the default MG5aMC reweighting module as applied to BSM reweighting specifically in the SM Effective Field Theory (SMEFT) using a SMEFTsim UFO model [23,24]. We finish the paper with a summary alongside discussion regarding possible future development directions and considerations for all three presented codes in section 5.

2 Rapid event extraction

The LHE file format is a human-readable XML-based format for storing parton-level hard scattering event information intended for interfacing between high energy physics (HEP) software in a generic yet simple way. However, despite the shared input/output (I/O) format, parton-level event generators have generally designed their own interfaces for matching their internal data formats to the read or written LHE file. The `Rapid event extraction` C++ library (`Rex`) is intended to serve as a simple and efficient tool for reading and writing LHE files, while providing a data-oriented internal data format with memory laid out specifically to simplify the matching between the natively object-oriented LHE format and modern event-level data-parallel programs[2].

To facilitate this goal, `Rex` has two internal storage formats which can easily be transposed between with a single function call: An OO LHE-adjacent tree structure where each event is stored as a separate `event` object, and a data-oriented SoA format of `process` objects where event data are merged into singular, contiguous arrays based on user-input sorting functions. While this latter structure was designed for the event-level data-parallel format of the `CUDACPP` plugin [12–17] for MadGraph5_aMC@NLO [21], we expect the functionality to be applicable to any current or future software looking to implement a data-parallel multi-event interface due to the necessity of data-oriented formats for proper utilisation of data-parallel hardware such as SIMD-enabled CPUs and SIMT GPUs.

Furthermore, `Rex` was designed with modularity in mind, both with respect to the necessary data within a given software and with respect to what file format the underlying event data is stored in on disk. As of version 1.0.0 `Rex` only has native support for the XML-based LHE v3.0 file

[2] The LHE format is intended for parton-level events generated from so-called *matrix element generators*; later stages of simulation additionally include parton showering and hadronisation of resulting hard scattering events and are typically stored in the HepMC format [25–27] which additionally preserves the topology of the interaction. HepMC events typically contain hundreds of particles, with the total number of particles per event varying in similar magnitude; a `Rex`-like algorithm where events are stored contiguously based on a well-defined yet generic measure of "equality" is thus uniquely well-suited for parton-level events and would be difficult to generalise to other aspects of HEP simulation.

format, but the internal data format and the I/O routines are completely disparate in implementation, yet simple to interface. We intend for this to make user-side extensions to other formats, such as e.g. the HDF5-based [28] LHEH5 format [29,30], minimal and possible with little user-side interfacing — the `lheReader` class for mapping any input data to the internal `Rex` data structures can be constructed from just two functions: one constructing `event` objects, and one constructing `initNode` objects. The former contain all event-level data necessitated by the LHE standard alongside generic storage for arbitrary software-dependent data, while the latter hold the overarching process information corresponding to the `<init>` node in the LHE standard. The `lheWriter`, mapping `lhe` objects to a user-provided output format, can be implemented similarly.

`Rex` was designed with the principal goal of simplifying efficient HEP software design while maintaining physics-driven data access with a generic base structure allowing both for the immediate use of `Rex` features and an extensive, adaptable interface for advanced use cases, all the while providing sufficient internal support for any level of complexity in between these extremes. The functionality of `Rex` can roughly be grouped into three categories based on these principles:

- **Physics-oriented data access:** LHE-based objects, including the `event`, `process`, and `lhe` structs, intended to give immediate access to physics data according to the LHE standard, with zero interfacing necessary other than that necessary to load an LHE file. These objects provide immediate access to all the underlying data by reference, such that they can be directly modified without needing to create and set data with additional function calls.
- **Helper classes for customisation:** Relatively simple wrappers for e.g. constructing event comparison functions, storing and accessing additional event-level information not part of the LHE standard, such as pdf information, or translation to and from other data formats. While they do not provide full customisability, these wrappers make it easy to set up more specific configurations than just immediate object-oriented event access.
- **Bare data and functionality:** Underlying fundamental data types and the templated base classes used to define them, as well as the non-wrapped function types the helper wrappers discussed above give access to, such as completely generic event sorters.

In the manual provided in section 2.1, these are presented in the order listed above, starting with the plug-and-play access to LHE format data in a C++ program without consideration for underlying data handling and ending with descriptions of the underlying data types and the definitions of interfacing functionality. Then, some simplified illustrative implementations and use cases are shown in section 2.2 in the same order.

2.1 Manual

This section is intended to provide a practical user manual for the `Rex` library, describing its structure and usage. Due to its intentional simple-to-complex and specific-to-generic design, this manual is split into three separate parts: the first, section 2.1.1, describes the default data access format for interfacing with LHE-style data using the `event`, `process`, and `lhe` types as well as loading and writing LHE standard files; the second section provides an introduction to the functionality wrappers allowing for customisation without the need for defining comparators, sorters, and type translators from scratch; finally, section 2.1.3 gives a description of the underlying data types and storage formats to allow power users full generic applicability of `Rex` with completely generic transposition between OO- and SoA-formats. For users just looking to use `Rex` to read and write the XML-based LHE format and simplify existing workflows, we recommend reading the first two parts, while power users looking to integrate a standardised data format for parton-level HEP information may also find interest in section 2.1.3.

2.1.1 Physics-driven data access

At its core, `Rex` is a library for accessing information according to the LHE standard while providing transposition between the OO-format given by the XML-based standard and a data-oriented SoA format where all event data is stored contiguously in memory, allowing for simple access to data parallelism using SIMD instructions and SIMT machines. This is increasingly important function with the rise of data-parallel hardware accelerators due to the preference for data contiguity of SIMD and SIMT hardware, which are typically designed to employ identical instructions across multiple simultaneous data flows. Consequently, the OO approach favoured in the last few decades can be a hindrance for the application of data parallelism due to the explicit separation of similar data into individual objects[3]. The three `Rex`-provided types relevant for the purpose of this interface are the OO `event` struct, the SoA `process` struct, and the overarching `lhe` struct.

[3] A small example: assume we have objects `P1`, `P2`, `P3` with different members on which we apply a series of functions `foo(Pn.one)`, `goo(Pn.two)`, `hoo(Pn.three)`. For data parallelism across function calls, it is not the `Pn` that need to be contiguous in memory but their member data, i.e. we need function calls of the form `foo_parallel({P1.one,P2.one,P3.one})` etc. Depending on the codebase, OO formats can make the implementation of SIMD- and SIMT-enabled code an arduous task.

Table 1 Event characterisation data as accessible through the `event` type in `Rex`. The `xxxUP` names correspond to the **u**ser **p**rocess naming convention for user processes in [31] with alternative names based on common terms used for said variables. All data are given by reference when accessed through the listed functions, meaning they can be modified directly through the access functions

Data	Type	Access functions
No. partons	`size_t`	`nUP()`, `n()`
Process index	`long int`	`idPrUP()`, `idPr()`
Event weight	`double`	`xWgtUP()`, `xWgt()`, `weight()`
Event scale	`double`	`scalUP()`, `scale()`
α_{EW}	`double`	`aQEDUP()`, `alphaQED()`, `aQED()`, `aEW()`, `alphaEW()`
α_S	`double`	`aQCDUP()`, `aQCD()`, `alphaS()`, `aS()`

Table 2 Parton-level data as accessible through the `event` type in `Rex`. The `xxxUP` names correspond to the **u**ser **p**rocess naming convention for user processes in [31] with alternative names based on common terms used for said variables. All data are given by reference when accessed through the listed functions, meaning they can be modified directly through the access functions. Additionally, views of individual parton lines are accessible through the `particle` member of the `event` type, which can also be accessed by the index operators `operator[](...)` and `at(...)` of `event` objects. The `particle` subtype has the exact same access functions as the `event` type, although accessing it through the former provides the data for a single event parton while the latter provides a vector of all values for each particle in the event

Data	Type	Access functions
PDG code	`long int`	`idUP()`, `id()`, `pdg()`
Status	`short int`	`iStUP()`, `iSt()`, `status()`
Mothers	`short int[2]`	`mothUP()`, `moth()`, `mother()`
Colour flow	`short int[2]`	`iColUP()`, `iCol()`, `icol()`
Momentum*	`double[4]`	`pUP()`, `momentum()`, `p()`, `momenta()`
Mass*	`double`	`mUP()`, `m()`, `mass()`
Lifetime	`double`	`vTimUP()`, `vTim()`, `vtim()`
Spin	`double`	`spinUP()`, `spin()`

*Note that while the LHE format stores particle momenta as arrays of 5 doubles ordered according to the (x, y, z, t, m) basis with parton mass m appended as a fifth and final entry, `Rex` stores momenta in the (t, x, y, z) basis and masses separately from the momenta

From an access perspective, the `event` type is quite simple — it contains all event-level data part of the LHE standard. Although internally these are stored with different names than typical, the `event` type has access function for many standard names for these variables, as detailed in tables 1 and 2 (internally, `Rex` uses custom types for arrays and vectors of arrays that ensure safety and contiguous storage — elaborated on in section 2.1.3 — but the important point to note is that e.g. vectors of four-momenta are stored contiguously and are accessed through a double index, the first referring to the line and the second to the momentum component; furthermore, `std::vector`s of the corresponding information can be accessed through the member function `flat_vector()`). Additionally, note that individual partons (given as `particle` objects) of an `event` object can be accessed through indexing access operators `operator[](...)` and `at(...)`. The `particle` type is a member of the `event` type, which gives a view of that particular index of the data stored in the `event` object, meaning `event`s can be treated either as collections of parton-level data or as collections of partons with individual data. Partons additionally have reference access to their individual momentum components through the functions `E()`, `px()`, `py()`, and `pz()`. When treating four-momenta, note that `Rex` stores these in the (t, x, y, z) basis with mass m stored separately, as opposed to the LHE format where they are treated as five-dimensional arrays in the basis (x, y, z, t, m).

Note that in contrast to common practice, data access in `Rex` is always done by reference unless otherwise specified. This has a two-fold purpose: one, it limits memory usage to a minimum by mitigating superfluous data copies unless explicitly requested by users and ensures memory contiguity when types are passed through several program layers; and two, it makes the creation, modification, and destruction of event-level data trivial. Both of these points are intended to simplify the utilisation of `Rex` for external software and analysis, especially when considering the extensive usage

of event-based sorting detailed below. Rather than having to explicitly map out the relationships between e.g. partons, events, subprocesses, and the full LHE container, reference access ensures that data modification properly propagates through the full hierarchy regardless of the level it is modified at: by applying a function to an `event` or a sorted `process`, all modifications are propagated to the `lhe` object which can be directly written to disk[4]. Beware that this can lead to harmful data modification, and as such we highly advise users to use `const` versions of access functions whenever write access is not explicitly necessary.

In addition to the direct data access provided by the functions mentioned above, `event`s can be provided arbitrary parton orderings through the member `std::vector<size_t> indices` storing a given parton ordering, and the `event_view` member type of `event`s accessible from the `event` member function `view()`, which overrides the indexing operator `operator[]` to index according to the `indices` vector (possibly ignoring partons stored in the owning `event`, depending on the ordering).

Additionally, individual particles can be stored in the `parton` struct, which is an owning but otherwise equivalent type to the `particle` struct. These can be used to create `event` objects or to append partons to existing `event`s using the `event(std::vector<parton>)` constructor or `add_particle(const parton&)` member functions, respectively. Both `parton`s and `particle`s have additional member functions to calculate observables, such as transverse momentum `pT()`, transverse energy `eT()`, transverse mass `mT()`, or longitudinal momentum `pL()` (each of which also have a corresponding squared operator `pT2()`, `eT2()`, `mT2()`, and `pL2()`), or azimuthal angle `phi()`, polar angle `theta()`, rapidity `rap()`, and pseudorapidity `eta()`. These functions calculate the derived quantities online, and can thus not be used to modify the underlying event data.

The `event` type comes with self-returning setters for each non-derived variable mentioned above, given by member functions `set_...`. These do not come overloaded with different names (although this could be implemented should it be desired), so exact names should be read from the header `Rex.h`. Events also have a vector `wgts_` which stores any additional event weights — relevant for `teaRex` reweighting — as well as a shared vector of weight labels `weight_ids` which is primarily meant to be accessed from an owning `lhe` object. Finally, specific scales $\mu_{F,R,PS}$ for factorisation, renormalisation, and parton showers, are provided, although they are not mandated by the LHE standard, and can either be accessed directly through the corresponding functions `muF()`, `muR()`, or `muPS()` (although these functions directly return a reference to the corresponding double which may be equal to 0), but can also be accessed through `get_... const` getters which first check whether the particular scale has value zero; if it is, these getters instead return `scalUP`.

Collections of events are transposed into the `process` type, which has all the corresponding members as an `event` object with the difference that every variable is stored in a contiguous vector, not just parton-level data. This includes the scales μ mentioned above. All these vectors can be accessed through identically named member functions as for `event`s. `process` objects additionally have access to a vector of shared pointers to `event`s, intended to be defined by `lhe` objects at transposition, and include (self-returning) transposition functions `transpose_...` for all contiguous vector quantities to map these quantities back into corresponding events without needing to transpose all data. These transposition functions are overloaded for all naming schemes shown in tables 1 and 2, as well as a total transposition operator `transpose()` which resets the vector of owned `event`s (without necessarily deleting existing `event`s, depending on scope) with new ones defined from the owned contiguous vectors. It is worth mentioning also here that the array-like types used to store `momenta`, `iCol`, and `mother` can be accessed as `std::vector`s of the corresponding type using the member function `flat_vector()`, which returns reference access to a vector of the data now lacking the double indices.

Both `event` and `process` objects have an additional `extra` member, which is an unordered map from labels given as `std::string` to the given value stored as `std::any` for `event`s and `std::vector<std::any>` for `process`es. These are detailed further in section 2.1.2. Finally, g_S can be calculated directly from α_S using member functions `gS()` which return a `double` for `event`s and `std::vector<double>` for `process`es.

Finally, the `lhe` struct is a wrapping type for both `event`s and `process`es, enabling transposition between the two formats. When only one of the two formats is loaded, transposition can be achieved through the member functions `transpose()`, and when both exist one can be used as basis to override the other with the overloaded member function `transpose(std::string source)` — or, to explicitly decide, one can call the member functions `events_to_processes()` or `processes_to_events()`. When transposing from `event`s to `process`es, the boolean member `filter_processes` determines

[4] As an example, in `teaRex` event weights are appended on a process-by-process basis without any consideration for the order of the corresponding events in the `lhe` container; without reference access such modifications would mean explicitly keeping track of event ordering at all stages of evaluation.

Table 3 Access functions for <init> node data in the LHE standard; the xxxUP names correspond to the **u**ser **p**rocess naming convention for user processes in [31] with alternative names based on common terms used for said variables. Just as in tables 1 and 2 these functions return the corresponding variable by reference, allowing them to be used both as setters and getters. Note that the process-specific information (xSec, xSecErr, xMax, and lProc) are properties of individual processes, an arbitrary amount of which can be stored in a single LHE file, and as such these variables are actually stored as std::vectors of the corresponding type, indexed according to the order they show up in the LHE file

Data	Type	Access functions
Beam IDs	long int[2]	idBmUP(), idBm()
Beam energies	double[2]	eBmUP(), eBm()
pdf group IDs	short int[2]	pdfGUP(), pdfG()
pdf set IDs	long int[2]	pdfSUP(), pdfS()
Weight scheme	short int	idWgtUP(), idWgt()
No. processes	unsigned short	nProcUP(), nProc()
Cross section(s)	double	xSecUP(), xSec()
Error(s)	double	xSecErrUP(), xSecErr()
Max weight(s)	double	xMaxUP(), xMax()
Process ID(s)	long int	lProcUP(), lProc()

whether to transpose directly from the event data or from the reordered and possibly filtered event_view.

By default, the lhe::transpose() function will sort events into individual subprocesses by their external partons, and will filter the data to that relevant to those external partons. The former can be changed by defining an event_hash_fn either using the eventSorter type detailed in section 2.1.2 or with a custom hash as discussed in section 2.1.3, while the latter can be changed directly by changing the boolean filter_processes member.

Additionally, the lhe type has a header member stored as std::any since the header itself is not defined in the LHE format — aside from the fact that reweighting information is stored in an <initrwgt> child node. Using the default reader, the header is stored as a shared pointer to a Rex format xmlNode, i.e. identically to the node in the read LHE file, allowing access and modification directly in the XML format. The <initrwgt> is also modified online when appending new weights to the lhe type, assuming that the stored header is of type std::shared_ptr<xmlNode>, meaning a rewritten reweighted LHE file will automatically account for new weights not just in the individual events but also in the header. Additionally, just like events and processes, lhe can store arbitrary information in the std::unordered_map<std::string,std::any> extra.

Besides event-level data stored in OO and SoA formats, the lhe types also stores process characterisation data according to the LHE standard. Like event characterisation data, this can be accessed by reference using several different access functions, as detailed in table 3.

Reading LHE files can be done through the free function load_lhef(), which is overloaded to take either std::istream objects or file paths given as std::strings as arguments, returning the loaded lhe object. Similarly, lhe objects can be written to disk using the free function write_lhef() which is similarly overloaded to write either to a std::ostream or to a file path given by an argument std::string, although for write_lhef() the first argument must be the loaded lhe object.

Besides LHE files, has simple support for the SLHA format for model parameters [31]. The slha class provides a simple dictionary-like storage container for named blocks of parameter types, with each parameter in each block defined by an int ID and a double value. These parameters can be accessed and modified through the functions:

```
double get(const std::string &block,
   int index, double fallback = 0.0);
void set(const std::string &block,
   int index, double value);
```

with fallback the value returned if the given parameter cannot be found. slha objects can be constructed from std::istreams or std::strings using the free functions to_slha(...), or loaded from disk with the free function load_slha(const std::string & filename). Note that Rex lacks support for named parameters, meaning all parameters must be defined by both a block name and an ID.

As a sidenote, as mentioned above Rex comes shipped with xmlDoc and xmlNode types, internally used for parsing the XML-based LHE format. The Rex XML parser was developed for two reasons:

1. Avoiding external dependencies; Rex and teaRex are intended to be entirely self-contained in order to ensure minimal issues when including them in other software, as well as avoiding long-term stability issues regarding different versions of other packages.
2. Optimised usage; generic XML parsers have very different design goals than what is needed for Rex, and consequently very different optimisation targets; the LHE format has well-defined conventions and minimal hierarchical structure, making generic XML parsing excessive and generally bloated[5].

Consequently, Rex provides a small, simple XML parser which may not adhere entirely to the full XML standard[6], but should a user want to interact with LHE files in this format (or XML files in general) it is possible — and of course, should users require more robust XML modification, a new XML interface to their preferred XML library can be written using the I/O read/write wrappers detailed in section 2.1.2. Unlike for the lhe type, Rex does not support direct file loading into the xmlNode format — instead, an XML file needs to be loaded into a std::string, after which it can be loaded into std::shared_ptr<xmlNode> using

```
std::shared_ptr<REX::xmlNode> node
   = REX::xmlNode::parse(raw);
```

at which point the XML file (or node) is accessible.

More technical details on the xmlNode type are provided in section 2.1.3; for the remainder of this section, we will limit ourselves to listing some of the relevant functionality of the class. First, XML node content can be read using the following member functions:

- std::string_view name(): Returns a view of exclusively the node name, excluding any attributes stored in the start tag.
- std::string_view content(): Returns a view of the full node (including children) *excluding* the start and end tags.
- std::string_view full(): Returns a view of the full node (including children) *including* the start and end tags.

XML attributes are stored as a minimal struct Attr with members Attr::name_view and Attr::value_view, both stored as string views and accessible through the member functions name() and value(), respectively. The attributes of an xmlNode can be accessed through the member function

```
const std::vector<Attr> &attrs();
```

which provides const read-only access. Attributes can be added and modified using the xmlNode member functions

```
void add_attr(std::string name_,
   std::string value_);
bool set_attr(std::string_view name_,
   std::string value_);
```

where the former adds a new attribute with the given name and value, while the latter sets an existing attribute to the new value. The returned bool from set_attr is true if successful and false if no attribute with the given name is found.

Children of an xmlNode are stored as a vector of (shared pointers to) xmlNodes. This vector can be accessed through the children() member function, although more extensive child treatment is possible using the following functions:

- void add_child(std::shared_ptr<xmlNode> child): Appends the given child to the end of the parent node.
- bool has_child(std::string_view name_): Checks if any children have a given name.
- std::shared_ptr<xmlNode> get_child (std::string_view name_): Returns the first child with the given name, returning a nullptr on failure to find any.
- std::vector<std::shared_ptr<xmlNode>> get_children(std::string_view name_): Returns a vector of all children with name name_.
- bool remove_child(...): Overloaded function which suppresses a given child from being written, returning true on success and false if the child could not be found. Argument can be either size_t giving the position in the vector of children, *xmlNode giving the address of the child, or std::string_view giving the name of the child. The final overload will only suppress the first child with the corresponding name, and thus has limited use for files with many identically named nodes.
- bool replace_child(size_t anchor, std::shared_ptr<xmlNode> child): Suppresses the child at the given position in the vector of children and replaces it for writing. Can also be called with std::string_view name instead of size_t, replacing the first child with the given name.

[5] Before the first internal XML parser was developed, several external XML parsers were used as placeholders. All tested ones either had issues regarding memory consumption or load speed, which the Rex parser overcomes due to assumptions about the LHE format.

[6] Rex should parse any LHE-compliant files without issue, although on write there may be formatting differences in data not defined in the LHE standard (e.g. number of significant digits written for different parameters). As of version 1.0.0, Rex LHE writing conforms to MadGraph5_aMC@NLO LHE formatting, with several unit tests to check that this formatting is perfectly preserved on write regardless of how the data is provided to the lhe object. However, if format customisation is requested by users, we could implement it.

Returns `true` on success and `false` on failure to find the given child.

This list is non-exhaustive, but should suffice for typical use cases. Full public functionality can be read from the header `Rex.h`.

2.1.2 *Functionality wrappers*

In order to support generic functionality without forcing users to create all beyond-default functionality from scratch, `Rex` comes with a plethora of methods for constructing custom versions of most functionality, such as comparators and sorters for `event`s, generic extra information for `event` and `lhe` objects, and generic writers and readers from and to the `lhe` data format.

Starting with the generic `extra` information stored in `event`s and `process`es, it is a member variable of type `std::unordered_map` which maps a name given as `std::string` to a value of type `std::any` (for `event`) or `std::vector<std::any>` (for `process`). For `process` objects this map needs to be accessed directly as a member variable `process.extra`, but the `event` type has templated access operators

```
template <typename T>
void set(const std::string&, T)
{...}
T &get(const std::string&)
{...}
const T &get(const std::string&) const
{...}
```

with the `get` functions internally handling the `std::any_cast<T&>` before returning the given object, as well as throwing a bad-any-cast error if called with the wrong type `T` or an out-of-range error if no element with the given name is found. To safely check whether `extra` has a given entry, the boolean member function `bool has(const std::string&)` will test existence without trying to access the entry. The `initNode` type, from which the `lhe` type inherits, has identical functionality for more generic non-event level information.

Sorting `event`s into `process`es in `Rex` is rather simple, although there are several types necessary to build up the sorting infrastructure. Although completely custom operators can be provided (as detailed in section 2.1.3), custom sorting schemes can also be created with built-in `Rex` routines by creating event comparators using the `eventComparatorConfig` type, which when combined with a set of events can create a boolean pass/fail filter for an input event using the `eventBelongs` type; by combining several `eventBelongs` objects a custom sorting/hash function can be created with the `eventSorter` type.

The `eventComparatorConfig` struct is made up of boolean `compare_...` members defining whether a particular trait of the LHE standard should be compared to determine whether two `event`s are "equal" or not – each of which comes with self-returning setters for all the access names provided for `event`s in tables 1 and 2 — as well as tolerances for how much all data of type `double` may differ (relatively) to determine equality. Additionally, a `std::set<int>` member `status_filter` allows defining the relevant values of `iStUP` for which partons to compare such that only partons whose status is included in `status_filter` are included for `event` comparisons and any whose status is omitted are ignored (unless it is empty, in which case no filtering is applied). Although tolerances for each `double` value are stored separately, no setters are defined for these other than the generic self-returning `set_tolerance(double)`, which sets all tolerances to the given value; should varied tolerances be required, consult the header `Rex.h` to see what these variables are named. The self-returning `set_status_filter`, however, is overloaded to support calls with `std::vector<int>`, `std::set<int>`, or any generic `Args...`; in the last case, it is assumed the arguments can trivially be converted into elements of `std::set<int>`.

Once an `eventComparatorConfig` has been customised, the resulting comparator can be accessed using the `make_comparator()` (()) member functions. This returns an `event_equal_fn`, a boolean function type which takes two `event`s as input and returns whether the `event`s are equivalent under the corresponding comparison, i.e. essentially a custom `operator==` (see section 2.1.3 for more details on local and global equivalence comparisons). The default `eventComparatorConfig` setup will compare all parton statuses, PDG codes, and masses, although the default `event` sorter only compares the PDG codes of external legs, creating `eventBelongs` objects for each set of external legs.

The `eventBelongs` type is simple: It is equipped with a vector of `event`s and an `event_equal_fn` function pointer (which can be set by users) and can test whether an input `event` matches any of its `event`s with respect to the comparator through the `belongs(event&)` member function, which can also be called through `operator()` (i.e. for an `eventBelongs eb` and an `event e`, `bool eb(e)`). This allows for relatively free-form pass/-fail tests on `event`s, and provides the basis for the default hashing structure used to sort `event`s in the `lhe` struct.

We explicitly define functions of `event`s returning bools as `event_bool_fn`s, which serve as generic pass/fail filters for individual events and the functional bridge between event comparators and event sorters, the intricacies of which we elaborate on in section 2.1.3. `eventBelongs` objects can emit their resulting `event_bool_fn` through the

`get_event_bool()` member function, creating a function pointer to its own `belongs` member. Then, sorting functions for `event`s are defined as

```
using event_hash_fn =
   std::function<size_t(event&)>
```

which essentially is any function mapping an `event` to `size_t`. Such hash functions can be created from `eventBelongs` objects with the `eventSorter` type, which is even simpler than the `eventBelongs` type: It consists of a vector of `eventBelongs` objects, and when its member function `position(event&)` is called it returns the first index to which the `event` the call to `eventBelongs::belongs` returns `true`. On a failure to map the `event` to any index, it returns `npos`. Similarly to how `eventBelongs` can provide functions for corresponding `event_bool_fn`s, the `eventSorter` struct has the member function `get_hash()` which returns a function pointer to its hashing function. The `lhe` type has a member variable `eventSorter sorter` which can be set using the self-returning setter `set_sorter` which is used at runtime to determine the splitting of owned `event`s when transposing to the `process` format. `lhe::set_sorter` can also be called with a generic `event_equal_fn` to automatically generate an `eventSorter` based on currently owned events.

With the internal treatment of LHE data described, I/O routines remain to be detailed. `Rex` comes equipped with two generic templated types `lheReader` and `lheWriter` which, as the names suggest, can be used to convert the `lhe` type to and from generic data formats. Although internally these types have some significant differences, for an end-user the experience is largely equivalent: The `lheReader` needs to be supplied with conversion functions from the template types `EventRaw` and `InitRaw` to the corresponding `Rex` types `event` and `initNode` — as well as optionally a function mapping a template `HeaderRaw` to `std::any` for the generic `lhe.header` object — and vice versa for `lheWriter`. For both classes, these translators are defined as

```
using InitTx = std::function
   <initNode(const InitRaw&>;
using EventTx = std::function
   <event(const EventRaw&)>;
using HeaderTx= std::function
   <std::any(const HeaderRaw&)>;
```

and the other way around for `lheWriter`. Here, `Tx` is used as a generic designation for "translator" functions, i.e. functions mapping one type onto another. Additionally, `EventTx` has surrounding helpers to support `std::function`s returning not only an `event` object, but also `std::shared_ptr<event>` or `std::unique_ptr<event>`.

These types can be constructed using the constructors

```
lheReader(initTx in_tx, EventTx ev_tx);
lheWriter(initTx in_tx, EventTx ev_tx);
```

with `HeaderTx` as an optional additional argument which will be ignored unless set, and we reiterate that the translator functions have opposite directionality for the two types. Alternatively, both types have self-returning setters `set_init_translator`, `set_event_translator`, and `set_header_translator`. `lheReader` has the member function `read`, which takes as input an `InitRaw` and an `EventRange`, as well as an optional `std::optional<HeaderRaw>`. `EventRange` must be an iterable object such that the following loop is well-defined:

```
for (const auto &er : events_raw)
   {
      evts.emplace_back(event_tx_(er));
      ...
   }
```

e.g. `std::vector<EventRaw>` or a similar type. Alternatively, the free function

```
lhe read_lhe(const InitRaw &in_raw,
   const EventRange &ev_raw,
   InitTx in_tx,
   EventTx ev_tx,
   std::optional<HeaderRaw>
      header_raw = std::nullopt,
   HeaderTx head_tx = nullopt)
```

can be called to handle all the details automatically and just return the resulting `lhe` object. Similarly, `lheWriter` has the `to_raw(const lhe &doc)` member function which returns an `lheRaw` object storing the resulting information (detailed below); alternatively, the free function

```
lheRaw write_lhe(const lhe &doc,
   InitTx in_tx,
   EventTx ev_tx,
   HeaderTx head_tx = nullopt)
```

can be called to handle the intricacies.

The `lheRaw` struct is a minimal storage container for raw LHE information, having only three members: `InitRaw init`, `std::vector<EventRaw> event`, and the optional `HeaderRawOpt header` (which must be of type `std::optional<...>`). As of `Rex` version 1.0.0 `lheReader` does not support reading using the templated `lheRaw` type, nor does it or `lheWriter` support the automatic I/O of generic types using function pointers for `std::istream` and `std::ostream`, but should there

be interest in such user-end simplifications, they could be implemented for future versions.

2.1.3 Fundamental types

Although the C++ `std::array` and `std::vector` types provide contiguous storage for fixed size (`array`) and dynamic size (`vector`) sequential type containers, as of the C++17 standard there is no container for dynamically sized containers of fixed size containers[7]. A container that supports multidimensional indexing while ensuring memory contiguity is especially important when trying to optimise HEP code, as many (and typically the most important) quantities are defined in terms of four-vectors and two-/four-spinors. This becomes especially important when optimising code for hardware acceleration using SIMD instructions or SIMT machines.

To treat this deficiency, three templated fundamental types are defined in `Rex`: fixed size arrays `arrN`, reference-like fixed size access objects `arrNRef`, and dynamically sized vectors of arrays `vecArrN`. These are templated with respect to the underlying object type `typename T` and the array dimensionality `size_t N` with explicit library-side instantiations for `T` $\in$ {`short, long, int, float, double`} and `N` $\in$ {2, 3, 4}. Additionally, aliases for `N` $\in$ {2, 3, 4} are provided:

```
using arr2<T> = arrN<T,2>;
using arr3<T> = arrN<T,3>;
using arr4<T> = arrN<T,4>;
```

and similarly for `arrNRef` and `vecArrN`. `arrN<T,N>` is a wrapper for a C-style array with type `T` and size `N`, with some additional functionality to circumvent the common pitfalls of using `T[N]`. However, `arrNRef<T,N>` is a non-owning proxy which can be accessed in the same manner as `arrN` — at its core, it is simply a pointer `T *q` referencing the `N` data starting at the location `*q` — allowing for `arrN`-like access to data stored in a separate container. Finally, `vecArrN<T,N>` is just `std::vector<T>` equipped with a custom iterator `nStrideIter<T,N>` such that e.g. `vecArrN<double,4>[0,1,2,...,M]` returns an `arrNRef<double,4>` object with `*q` pointing to the underlying element at `std::vector<double>[0,4,8,...,4×M]` etc. Similarly, operators relating to the size of a `vecArrN` object are multiplied and divided by `N` for access, reservation, sizes, and so on. However, the underlying vector can also be accessed by reference with the method `vecArrN::flat_vector()`, which is e.g. how `process` momenta are passed to the scattering amplitude routines in MadtRex (detailed further in section 4).

Note that `arrN` and its derived types are designed for trivial SIMD alignment specifically for real numbers – while one can construct an `arrN<std::complex>`, the interweaved real and imaginary parts make it unsuited for SIMD operations. However, it would be relatively simple to write a complex-like class `cArrN<T,N>` consisting of two separate `arrN<T,N>` objects corresponding to the real and imaginary parts of the array with explicitly defined elementary operations `+,-,*,/`. While such a class is not provided with `Rex` version 1.0.0, it could be implemented in a future version alongside derived `cArrNRef<T,N>` `vecCArrN<T,N>` types should there be interest for it.

Furthermore, `arrN` and `vecArrN` only provide 1- and 2-dimensional storage and access, making matrix and tensor multiplication non-trivial. We do note that that `arrN` has a method `arrN::dot(const arrN& other)` allowing for generic Euclidean dot products between `arrN` objects. This can be used to implement matrix multiplication between `arrN` and `vecArrN` objects, but again, such developments are left for future work.

Although we leave the description of the `event` and `process` types for later, we note here that `arrN` and derived types are used to store not just momenta (`arr4<double>`, but also particle mothers (`arr2<short int>`) and colour flow (`arr2<short int>`)[8].

Aside from the storage types, there are some function type aliases that are relevant for in-depth `Rex` usage. The first are event comparator types:

```
using event_equal_fn =
   std::function<bool(event &, event &)>;
using cevent_equal_fn =
  std::function<bool
  (const event &, const event &)>;
```

which can be used as generic equality comparators for event objects and serve as the foundation of the boolean `eventBelongs` event testers and hashing `eventSorter` event sorters. The reason for both mutable and const versions of this type is simple: Mutable comparators can sort event partons online while doing the comparison. While this feature is not used inside `Rex`, it could be used to optimise the process of sorting `event`s into individual `process`es. The default `operator==` for `event` objects exclusively compares the (unordered) PDG codes of the external legs, but this can be changed globally using the function

```
void set_event_comparator
```

[7] While e.g. a vector of arrays necessitates the arrays to be contiguous and the elements of the arrays to be contiguous, the elements of sequential arrays are not necessarily adjacent in memory as the arrays may have start and end padding. While the `std::mdspan` introduced in C++23 does not have this restriction, it is not yet supported by most compilers.

[8] `arrN` is also used to store beam IDs, beam energies, pdf groups, and pdf sets, but here there is no particular reason to prefer `arrN` over `std::array`.

```
    (cevent_equal_fn fn);
```

Internally, the operator== function has access to a std::shared_ptr<cevent_equal_fn> which shares scope with a std::mutex that is only accessible from operator== and set_event_comparator. Note that the global comparator must be const, while local comparators (e.g. those used in eventBelongs) may modify the compared events.

The other two function types relevant for low-level configuration are

```
using event_bool_fn =
   std::function<bool(event &)>;
using event_hash_fn =
   std::function<size_t(event &)>;
```

both of which also have corresponding const cevent_... types. The former is a generic pass/fail type for event objects, although internally in Rex it serves no purpose other than acting as an intermediate type between the comparators above and the hash type just shown,[9]. The event_hash_fn type, however, is a fundamental part of Rex, as it is used when transposing lhe objects between the event and process formats — the lhe struct has an event_hash_fn member, and before creating the process objects all events are sorted based on the indexing provided by this hash function. Specifically, each unique hash value will be mapped to a unique process ordered as per the order the events appear in the lhe data[10], with the caveat that events with hash REX::npos = (size_t)-1 will be mapped to the very last process corresponding to "unsorted" events. Custom hashes can be provided to the lhe object using the self-returning member function set_hash(event_hash_fn hash), where we note that cevent_hash_fn can be automatically converted to event_hash_fn (although the opposite conversion is impossible).

One final intricacy beyond the scope of typical usage is the XML parser supplied with Rex, although the specifics here are unlikely to be interesting for power users. To give a brief description, the xmlNode class is a tree-like structure of pointers between mother nodes and children with a shared loaded data storage in std::string format, to which all the individual nodes only have access through std::string_views of their respective data. Children can be added, removed, and replaced, with new nodes (or node attributes) owned by the child in question, while any data left unmodified is kept as std::string_views of the original std::string.

We omit more extensive details on XML treatment as Rex is not primarily intended to for XML utility, although we reiterate that Rex XML handling is designed with respect to primarily reading and secondarily writing LHE format files and that it may not perfectly fulfil the XML standard when handling generic XML files nor be particularly efficient in handling more complex node hierarchies than the almost linear LHE format.

2.2 Use case illustrations

In this section, we intend to illustrate some uses for the functionality mentioned above to show how simple Rex is to use for writing new software or for unifying a format for existing software. Starting with some elementary uses, LHE files can be read, sorted and transposed, and written by

```
REX::lhe file = REX::load_lhef(inpath);
file.transpose();
std::ofstream out(outpath);
file.print(out);
```

where the default event sorting algorithm was used, comparing the PDG codes of external partons. Alternatively, an illustration of the more generic comparison capabilities is shown in algorithm 1. Rex internally always uses explicitly defined comparison operators, leaving users free to define and utilise them however they want within their codebase — of course, noting that other types need their local event_bool_fns defined as well. Note that the function externalComp shown in algorithm 1 is equivalent to the default event comparison operators used throughout Rex.

With the simplicity of constructing event comparators illustrated, it follows that it is just as simple to create the boolean pass/fail tests provided by the REX::eventBelongs type:

```
std::vector<REX::event> es = ...
event_equal_fn cmp = ...
auto belong = REX::eventBelongs(es,cmp);
```

and with that we can test whether a given event fits our particular conditions formulated in terms of other events and specific fields of comparison with these. This immediately extends to the creation of event_hash_fns using the REX::eventSorter type as

[9] Within Rex event_bool_fns simplify the creation of hashes as the latter can be constructed from sets of the former, automatically creating a coherent indexing with respect to the order of the event_bool_fns. More generally, they can be used to filter particular process types for analysis or additional processing.

[10] I.e. if we have unique hashes 0 and 1, the process objects may be ordered with events corresponding to hash 1 first if the first event object has hash 1. This avoids segmentation faults when sorting processes, but means custom hash functions may not end up corresponding to the ordering of the resulting processes.

Algorithm 1 Illustration of event comparisons using the `eventComparatorConfig` type to easily create generic comparison operators which can either be called locally or set for global comparisons.

```
REX::eventComparatorConfig comp1,comp2;
// Comparators for all external legs and only final-state particles
comp1.set_status_filter({-1,1}).set_pdg(true).set_mass(true);
comp2.set_status_filter({1}).set_pdg(true).set_mass(true);
auto exeternalComp = comp1.make_const_comparator();
auto finalComp = comp2.make_const_comparator();

REX::event ev1, ev2;
ev1.set_n(4).set_pdg({21,21,6,-6}).set_status({-1,-1,1,1}).set_mass({0,0,173,173});
ev2.set_n(4).set_pdg({2,-2,6,-6}).set_status({-1,-1,1,1}).set_mass({0,0,173,173});
assert( !externalComp(ev1,ev2) );
assert( finalComp(ev1,ev2) );

// Changing global comparators
REX::set_event_comparator( externalComp );
assert( !(ev1 == ev2) );
REX::set_event_comparator( finalComp );
assert( ev1 == ev2 );
```

```
std::vector<REX::eventBelongs> belong =
   ...
REX::eventSorter hash(belong);
```

and an `event` can now be hashed through the member function `REX::eventSorter::position`, and vectors of `event`s can be hashed at once using the member function `REX::eventSorter::sort`. Note that a failed hash will return `REX::npos`.

Of course, `eventSorter`s can also be constructed from generic `(c)event_bool_fn`s, and similarly, the `event_hash_fn` used when transposing `event`s in a `REX::lhe` object to the `REX::process` type can be set explicitly using the `lhe::set_hash` member function for more generic uses, but the simpler application of `eventBelongs` objects should suffice for most use cases.

One additional note for event sorting is the ordering of `particle`s in an `event`. As mentioned above, individual `particle`s are given as views of the corresponding data row stored in an `event`, and these can either be accessed directly from the `event` — in which case the `particle`s are ordered according to the underlying data storage — or through a `REX::event_view`, which masks the `REX::event::operator[]` through the vector `REX::event::indices`, i.e. (somewhat simplified)

```
particle event_view::operator[](size_t i)
{ return this->evt[indices[i]]; }
```

where `indices` will default to `{0,1,...}` unless explicitly set beforehand. Setting these indices is done through the `event` member function `set_indices`, which is overloaded to take as input either a vector of `size_t`s, *or* another event to be indexed with respect to[11]. While for most use cases `event::indices` need to be set explicitly, `Rex` has one access point where they are set automatically: When calling `eventBelongs::belongs` with a non-constant `event`, if the `event` succeeds its `set_indices` member function will be called with the `event` it was compared to:

```
bool eventBelongs::belongs(event &e){
   ...
   for(auto ev : this->events){
      if(this->comparator(*ev,e)){
         e.set_indices(*ev);
         return true;
      }
   }
   return false;
}
```

which is how `lhe` objects are sorted by default, which, when combined with the default member setting `filter_processes = true` means transposed `process` objects unless otherwise specified will be filtered to the particles used for event comparison and each `event`'s data ordered with respect to the events used in sorting. This particular feature is important for consideration when using the `process` type as input to (data-parallel) functions where it is assumed that each contiguous data set is ordered iden-

[11] By indexing an `event` (`orig`) with respect to another (`oth`), we mean setting the indices such that when `orig` is accessed through the `event_view`, `particle`s will have the same order as they would in the data structure of `oth`, the event we indexed with respect to. This indexing only considers the PDG codes and statuses of partons, and will ignore partons that do not appear in `oth`, i.e. their position will not be included in the vector of indices.

Algorithm 2 Usage illustration for the `REX::lheReader` and `REX::lheWriter` types, using function pointers to the corresponding translators `xml_to_TYPE` for types `REX::initNode`, `REX::event`, and `std::any` (with `std::any` used as a generic container for the `<header>` node in the LHE format), and vice versa for translators `TYPE_to_xml`.

```
using xmlReader = lheReader<
  std::shared_ptr<xmlNode>,
  std::shared_ptr<xmlNode>,
  std::shared_ptr<xmlNode>
>;

using xmlWriter = lheWriter<
  std::shared_ptr<xmlNode>,
  std::shared_ptr<xmlNode>,
  std::optional<std::shared_ptr<xmlNode>>
>;

using xmlRaw = lheRaw<
  std::shared_ptr<xmlNode>,
  std::shared_ptr<xmlNode>,
  std::optional<std::shared_ptr<xmlNode>>
>;

const xmlReader &xml_reader(){
   static const xmlReader b{&xml_to_init,
      &xml_to_event, &xml_to_any};
   return b;
}

const xmlWriter &xml_writer(){
   static const xmlWriter t{&init_to_xml,
      &event_to_xml, &header_to_xml};
return t;
}

xmlRaw to_xml_raw(const lhe &doc){
   return xml_writer().to_raw(doc);
}
```

tically, which is assumed and used in MadtRex (elaborated on in section 4).

The final functionality important to consider is support for generic I/O, using wrappers for "translator functions" between the `event` and `initNode` formats and some arbitrary alternate types `InitRaw` and `EventRaw`. Of course, it is entirely possible to create a generic `REX::lhe` constructor from an arbitrary data format, but our intention here is to provide a minimal constructor for any arbitrary data format.

The templated `REX::lheReader` and `REX::lheWriter` classes are from a user-side perspective similar — they both use two to three "translator" function pointers in order to construct a translator to-or-from the `REX::lhe` type and a generic input/output format.

First, we consider `lheReader`. It has three template arguments,

```
template <class InitRaw, class EventRaw,
   class HeaderRaw = std::monostate>
   class lheReader{
   public:
   using InitTx = std::function<initNode
      (const InitRaw &)>;
   using EventTx = std::function<
      std::shared_ptr<event>(
         const EventRaw &)>;
   using HeaderTx = std::function<
   std::any(const HeaderRaw &)>;
...}
```

where the `HeaderRaw` type is not necessary but allows for handling of generic data containers for LHE headers. The `lheReader` type has two explicit constructors:

```
explicit lheReader(InitTx init_tx,
   EventTx event_tx)
{ set_init_translator(
     std::move(init_tx));
  set_event_translator(
     std::move(event_tx));
}
```

which only treats the mandatory translators for the `event` and `initNode` types. The second constructor additionally handles an optional translator from a generic `HeaderRaw` to the `std::any` type used to store the `<header>` LHE node:

```
template <class InitTx,
   class EventTx, class HeaderTx>
lheReader(InitTx init_tx,
   EventTx event_tx, HeaderTx header_tx)
{  set_init_translator(
       std::move(init_tx));
   set_event_translator(
       std::move(event_tx));
   set_header_translator(
      std::move(header_tx));
}
```

i.e. `lheReader`s need translators for `event`s and `initNode`s, and optionally also `header`s, where the resulting type for `header` is `std::any`. Note that while `EventTx` is defined in terms of the type `std::shared_ptr<REX::event>` there are surrounding helpers converting raw `event`s or `std::unique_ptr`s to `event`s to the `std::shared_ptr` format used in `lhe`, meaning the user-provided `EventTx` does not need to provide a shared pointer directly (although we of course suggest using shared pointers where applicable to ensure object continuity). Alternatively, `lheReader` like most `Rex` types has self-returning setters `set_x_translator`, for $x \in$

{init,event,header}. Or, to minimise type interfacing, the templated free function read_lhe can be used:

```
template <class InitRaw, class EventRange,
  class HeaderRaw = std::monostate,
  class InitTx, class EventTx,
  class HeaderTx = std::nullptr_t>
lhe read_lhe(const InitRaw &init_raw,
  const EventRange &events_raw,
  InitTx init_tx, EventTx event_tx,
  std::optional<HeaderRaw> header_raw =
     std::nullopt,
  HeaderTx header_tx = nullptr,
  bool filter_processes = false)
{using EventRawT = typename std::decay<
   decltype(*std::begin(
   events_raw))>::type;
 lheReader<InitRaw, EventRawT, HeaderRaw>
   b;
 ...
 return b.read(
   init_raw, events_raw, header_raw);
}
```

which handles all the intricacies of lhe construction, only necessitating the relevant translators and object sets to construct lhe objects. The lheWriter type is similar, with the caveat that it has an intermediate return type lheRaw to store initNode data, event data, and optionally header data,

```
template <class InitRaw, class EventRaw,
   class HeaderRawOpt>
struct lheRaw
{
   InitRaw init;
   std::vector<EventRaw> events;
   HeaderRawOpt header;
};
```

and is just a minimal storage container for the raw data (assuming that events are stored in an object-oriented format).

With this in mind, lheWriter is defined almost identically to lheReader,

```
template <
   class InitRaw, class EventRaw,
   class HeaderRaw = std::monostate>
class lheWriter
{  public:
   using InitTx = std::function<InitRaw(
     const initNode &)>;
   using EventTx = std::function<
     EventRaw(event &)>;
   using HeaderTx = std::function<
     HeaderRaw(const std::any &)>;
   using result_t = lheRaw<
     InitRaw, EventRaw, HeaderRaw>;
```

with constructor

```
  lheWriter(InitTx init_tx,
     EventTx event_tx,
     HeaderTx header_tx = HeaderTx{})
   : init_fn_(std::move(init_tx)),
     event_fn_(std::move(event_tx)),
     header_fn_(std::move(header_tx)) {}
...}
```

and identically to lheWriter it has self-returning setters set_init_translator,set_event_translator, as well as an optional set_header_translator. And, again, there is the free function write_lhe, although this one has the form

```
template <class InitRaw,
  class EventRange,
  class EventRawT = typename
  std::decay<decltype(*std::begin(
    std::declval<EventRange>()))>::type,
  class HeaderRaw = std::monostate,
  class InitTx, class EventTx,
  class HeaderTx = std::nullptr_t>
lheRaw<InitRaw, EventRawT, HeaderRaw>
  write_lhe(lhe &doc,
    InitTx init_tx,
    EventTx event_tx,
    HeaderTx header_tx = nullptr)
```

where the only significant change to read_lhe is that the leading argument is now a singular REX::lhe object rather than all the raw objects. Although Rex internally uses direct conversions from singular XML nodes to Rex types and writes directly to std::ostream, it comes shipped with implementations for the REX::xmlNode format to provide illustrations. For reference, these are shown in algorithm 2, where the exact usage of these types are provided for the xmlNode types without details regarding the internal structure of the xmlNode itself. Essentially, given a function that turns e.g. a generic EventRaw into a REX::event, a translator can trivially be constructed and called using this function and the source objects.

2.3 Benchmarks

While it is difficult to profile the full extent of a wide-ranging library such as Rex, we can still benchmark some of its standard functionality. For the purpose of providing a reasonable showcase of what we expect to be typical usecases for Rex, we here present the event throughput of some standard functionality for a generic workflow: throughputs (in terms of events per second) for reading and sorting LHE files. We will analyse this using the standard XML-based LHE file

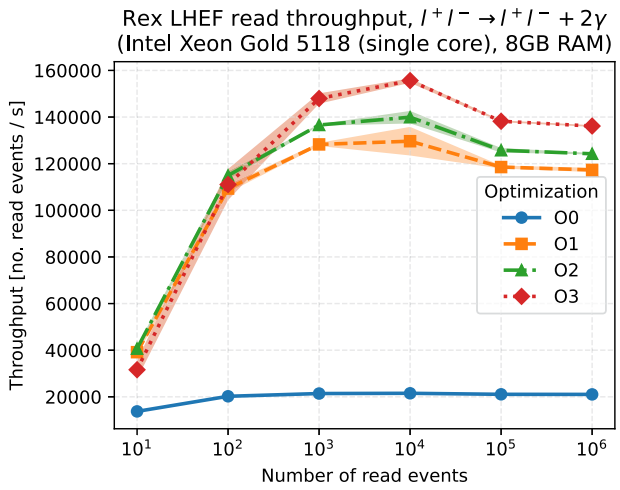

Fig. 1 Read throughput for an electroweak LHE sample using `Rex` as a function of file size in terms of number of events for optimisation levels `-O0` through `-O3` using `g++` version 13.2.0. Each point gives an average throughput from 100 measurements, with standard deviations highlighted. To ensure only the library functionality itself was measured, the benchmark executable was compiled with no optimisations. It is clear that the most significant speed-up comes from `-O1` optimisation, although `-O2` and particularly `-O3` do provide additional load speed-up (compare with Fig. 2, where `-O3` has no significant speed-up compared to `-O2`.) The dip at 100 000 events coincides with the corresponding LHE file exceeding the 16.5 MB L3 cache of the Intel Xeon Gold 5118 [32]

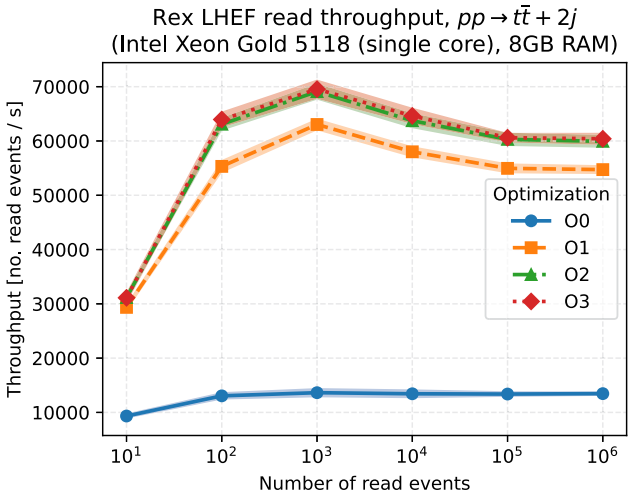

Fig. 2 Read throughput for a QCD LHE sample using `Rex` as a function of file size in terms of number of events for optimisation levels `-O0` through `-O3` using `g++` version 13.2.0. Each point gives an average throughput from 100 measurements, with standard deviations highlighted. To ensure only the library functionality itself was measured, the benchmark executable was compiled with no optimisations. It is clear that the most significant speed-up comes from `-O1` optimisation, although `-O2` does provide additional load speed-up while `-O3` does not appear to have any significant impact (compare with Fig. 1, where `-O3` has significant speed-up compared to `-O2`.) Although the throughput dip is more gradual than in Fig. 1, the plateau is once again reached at 100 000 events, which also for these samples is when the LHE file size exceeds the 16.5 MB L3 cache of the Intel Xeon Gold 5118 [32]

format — which `Rex` comes shipped with parsers for — to provide a benchmark for `Rex`' efficiency.

Starting with read throughput: since the XML-based default format in `Rex` needs to parse any additional node data, we will test both electroweak samples (whose events contain only information belonging to the LHE standard) and QCD samples (whose events may contain additional information regarding e.g., the event-specific pdf scale) generated with MG5aMC. These measurements are provided in Figs. 1 and 2, where the function `REX::load_lhef` was timed in total 100 times for each file for each tested level of compiler optimisation (from none with `-O0` to maximal with `-O3`) applied to the `Rex` library compilation. For all tests, the benchmark executable was compiled without optimisation to ensure times were representative of only `Rex` functionality.

Figures 1 and 2 provide several interesting insights, especially when compared. First and foremost, at least `-O1` optimisation is necessary for `Rex` to reach its potential — for both sample sets, `-O1` provides a roughly factor 6 speed-up compared to no compiler optimisation, with `-O3` only being marginally faster than `-O1`. Additionally, for both samples a throughput plateau is reached at the samples with 10^5 events, which for both sets coincides with the size of the loaded LHE file exceeding the 16.5 MB L3 cache of the Intel Xeon Gold 5118 tests were run on at 86.5 MB and 110.0 MB, respectively[12]. More notable are the significantly different throughputs measured in Figs. 1 and 2: already at 100 events, the EW sample is read twice as fast as the QCD sample. As far as `Rex` is concerned, the only significant difference between these files is the additional `<mgrwt>` child node each event has, storing renormalisation and pdf information. The likeliest source of slowdown are the string operations in appending these children to the member `REX::event::extra` (detailed in section 2.1.2), although more extensive tests are left for future development.

Next, we turn to event sorting. Specifically, for an already loaded `REX::lhe` we measure the runtime of the function call

```
lhefile.transpose();
```

i.e. the time taken to both sort the `REX::event` members and then extract their information into `REX::process` objects. Furthermore, we use the default sorting method

[12] `REX::load_lhef` streams the loaded file rather than reading it directly into memory, but native data types are only marginally smaller than the plaintext LHE format. A back-of-the-envelope calculation suggests `event`s with only data given by the LHE standard should at most be $\sim$ 50% smaller than the corresponding plaintext, excluding any padding or other surrounding infrastructure. Some preliminary tests on ΔRSS suggest that a `REX::lhe` object is similarly sized to the corresponding LHE file, reinforcing this conclusion.

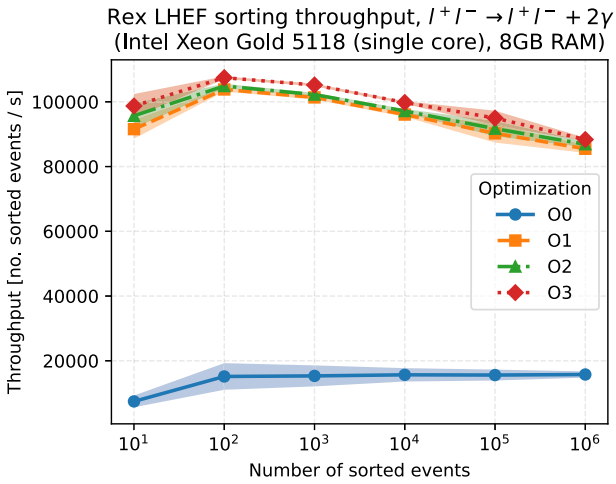

Fig. 3 Read throughput for an electroweak LHE sample using `Rex` as a function of file size in terms of number of events for optimisation levels `-O0` through `-O3` using `g++` version 13.2.0. Each point gives an average throughput from 100 measurements, with standard deviations highlighted. To ensure only the library functionality itself was measured, the benchmark executable was compiled with no optimisations. It is clear that the most significant speed-up comes from `-O1` optimisation

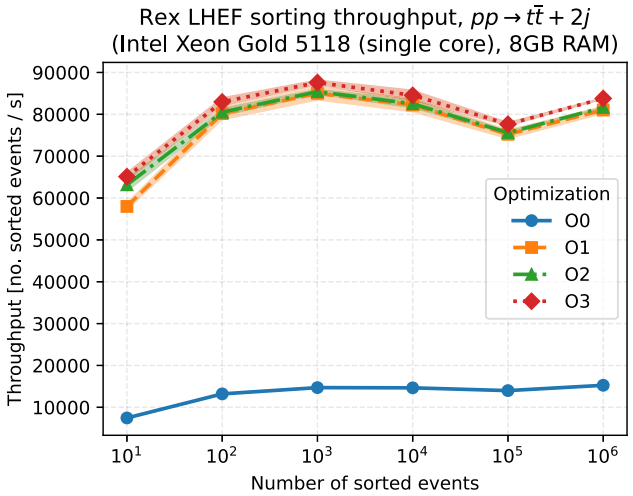

Fig. 4 Read throughput for a QCD LHE sample using `Rex` as a function of file size in terms of number of events for optimisation levels `-O0` through `-O3` using `g++` version 13.2.0. Each point gives an average throughput from 100 measurements, with standard deviations highlighted. To ensure only the library functionality itself was measured, the benchmark executable was compiled with no optimisations. It is clear that the most significant speed-up comes from `-O1` optimisation

where the `REX::lhe` object creates a `REX::eventSorter` online, comparing external partons of events and appending the current external legs to the sorter if the sorter does not recognise this particular configuration. The measured throughputs, again taken as the mean of 100 measurements with standard deviations highlighted, are shown in Figs. 3 and 4.

Once any initial overhead is overcome, the expected complexity scaling of sorting LHE files (just as for reading) is $\mathcal{O}(\#\text{events})$ which should result in a roughly constant event throughput. This aligns well with Figs. 3 and 4. However, there are some interesting points of consideration: first, in comparison to Figs. 1 and 2 almost all compiler optimisation here comes from `-O1` optimisation, with no notable difference between `-O1` and `-O2` past the minimal samples with 10 events; additionally, the differences in throughput between the sample sets are small. One final observation is the different behaviour between the sample sets as functions of the number of events: the EW sample has its peak throughput early, before decreasing slightly with increasing sample sizes; on the other hand, the QCD sample has a throughput increase between the samples with 10^5 and 10^6 events.

The differences between Figs. 3 and 4, and Figs. 1 and 2 are unsurprising: unlike read, which involves file streaming, string manipulation, type conversions etc., `REX::eventBelongs`-based sorting only involves object comparisons and copies, leaving little to optimise beyond memory management and function call ordering. Furthermore, the read overhead seen for QCD samples in Fig. 2 is driven by the additional string manipulation when loading events' XML child nodes — by the time sorting occurs, these are already stored as `std::shared_ptr<REX::xmlNode>`s owned by their events and when transposing to the `REX::process` format only a pointer is copied rather than the full string content, making the overhead between EW and QCD samples far less significant in sorting than for reading.

One final point of interest is the differing behaviour between Figs. 3 and 4. Although we do not know the origin of the increased throughput going to the million events sample in Fig. 4, it is not surprising that the two figures behave slightly differently. To see this, let us consider the complexity scaling of event sorting more in-depth: we have noted that we should see complexity scaling as $\sim \mathcal{O}(\#\text{events})$ (i.e. throughput should change minimally as the number of events per sample increases), but the leading constant comes from several origins. In particular, two:

$$t_{\text{total}} \simeq t_{\text{sort}} + t_{\text{transpose}}, \tag{1}$$

i.e. the two runtime costs are the event hashing and the transposition of event data to `REX::process` objects, both of which clearly should be linear in time. However, recall the default hashing mechanism for `REX::lhe` objects: events are sorted into groups based on their unordered external legs, with initial- and final-state particles separated. Particularly, this means that the time taken when sorting a sample depends on the number of distinct parton configurations within that sample, even if the following transposition will differ minimally in runtime. For reference, the samples used for Fig. 3 have 6 distinct parton configurations, all of which are sam-

pled already for the 100-event sample. On the other hand, the samples in Fig. 4 have 65 distinct parton configurations which in the unweighted samples tested are sampled at very different rates — in fact, only the 10^6-event sample actually contains all 65 configurations. With this in mind, differing throughputs in Fig. 4 may just be luck with respect to the ordering of events in the LHE file; if the file happens to start with less common configurations, the dominant ones will have to go through more comparisons before having their hash determined.

3 Tensorial event adaption

The `tensorial event adaption with Rex` library — `teaRex` — is an extension to `Rex` adding support for completely generic parton-level event reweighting using the SIMD- and SIMT-friendly `process` data format for sets of events. Simply, `teaRex` adds a `reweightor` type inheriting from the `REX::lhe` type which in addition to `lhe` members also owns a vector of `procReweightor`s; a type defined to perform event-level reweighting for singular `process` objects and automatically appending them to the `event`s owned by the relevant `reweightor` type object. Additionally, `teaRex` comes shipped with support for so-called "matrix element reweighting" where a given event is reweighted to different values of model parameters assuming model parameters are stored as SLHA cards on disk and scattering amplitudes can be evaluated from `REX::process` objects. Further details are provided in section 4, which primarily uses this implementation for its SIMD- and SIMT-enabled leading order reweighting.

For reference, parton-level event weights in HEP are defined by [22]

$$w = f_1(x_1, \mu_F) f_2(x_2, mu_F) \ |\mathcal{M}|^2 \ \Omega_{PS}, \tag{2}$$

with f_i the pdf evaluated at Bjorken fraction x_i and factorisation scale μ_F, $|\mathcal{M}|^2$ the absolute scattering amplitude squared of the event at the given phase space point, and Ω_{PS} the phase space measure of the corresponding event. Notably, these contributions all factorise, meaning that evaluating the resulting event weights from changing one of them can be calculated without having to re-evaluate the others. As an example, consider parameter reweighting where some physics model parameter entering the scattering amplitude $\mathcal{M}$ is changed, yielding

$$w' = f_1(x_1, \mu_F) f_2(x_2, mu_F) \ \left|\mathcal{M}'\right|^2 \ \Omega_{PS} \tag{3}$$

$$= \left(\prod_{i=1,2} f_i(x_i, \mu_F) \right) \ |\mathcal{M}|^2 \ \frac{\left|\mathcal{M}'\right|^2}{|\mathcal{M}'|^2} \ \Omega_{PS} \tag{4}$$

$$= \frac{\left|\mathcal{M}'\right|^2}{|\mathcal{M}|^2} w, \tag{5}$$

i.e. weights in a new model can be defined in terms of weights in the original one as long as the new model weights are non-zero only in a subspace of the phase space of the original model. Since different stages of HEP event simulation generally factorise (e.g. hard scattering events are disparate from hadronisation are disparate from detector simulation) this implies event samples simulated in one model can be repurposed for a different one under certain mathematical restrictions by only re-evaluating the scattering amplitude of the underlying hard scattering process.

A user manual for `teaRex` is provided in section 3.1 — which gives extensive details for how to use `teaRex` for parton-level event reweighting — and some implementation use case codes are shown in section 3.2.

3.1 Manual

`teaRex` is a small extension to `Rex` which adds types for appending new event weights to existing `lhe` objects using completely generic "reweighting functions" (`weightor`s), by which we mean any function acting on a `process` object and returning a (shared pointer to a) vector of `double`s;

```
using weightor = std::function<
  std::shared_ptr<std::vector<double>>
  (process &)>;
```

which are intended to evaluate (arbitrary) resulting weights for all events stored in a `process` object. If reweighting parameters are global, `teaRex` also supports "reweighting iterators" run between calls to `weightor`s and can thus safely modify global data without impeding the `weightor` calls,

```
using iterator = std::function<bool()>;
```

which can e.g. be used to reweight an entire event set for differing model parameters as is done in MadtRex (cf. Sect. 4).

This manual will be split into three sections: Sects. 3.1.1 and 3.1.2 illustrate how to use `teaRex` functionality to implement event reweighting using the user-friendly helper functions provided by `Rex` as well as completely generally, respectively; then Sect. 3.1.3 showcases the specific SLHA parameter reweighting implementation shipped with `teaRex` for use in MadtRex.

3.1.1 Default usage

The primary functionality of `teaRex` is provided by the `procReweightor` and `reweightor` types — plus the helper `threadPool` type managing multithreading

between individual processes — which handle the LHE-level handling of weight information treatment when reweighting events. The key assumptions made for teaRex are:

- Each process object is exclusively well-defined, i.e., no event belongs to multiple subprocesses.
- Individual processes are independent algorithmically and can be reweighted simultaneously.
- Reweighting routines are symmetric with respect to particle ordering[13].

Starting with the procReweightor struct, it is an independent type not inheriting from any other types, primarily to avoid double-counting information (as the reweightor type inherits from the REX::lhe type). Simply put, it has a boolean pass/fail filter to determine whether events belong to its corresponding process (given by a REX::event_bool_fn or REX::eventBelongs object), a weightor normaliser defining the original weights of its events, a vector of weightors to perform the relevant reweighting, and a shared pointer to its corresponding process which is intended to be simultaneously owned by the procReweightor and the surrounding reweightor. The procReweightor type has constructors

```
procReweightor(weightor rwgt_fn);
procReweightor(weightor rwgt_fn,
   REX::eventBelongs selector);
procReweightor(
   std::vector<weightor> rwgts);
procReweightor(std::vector<weightor> rws,
   REX::eventBelongs selector);
procReweightor(std::vector<weightor> rws,
   REX::eventBelongs selector,
   weightor normaliser);
```

where each constructor including eventBelongs has an overloaded corresponding constructor using std::shared_ptr<REX::eventBelongs> which we advise users to prefer when plausible,[14]. Note that the order of parton-level variables by default are sorted according to the order of said particles in the events making up the eventBelongs object when transposing from the event to the process format, as shown in section 2.2. Alternatively, instead of an eventBelongs object an arbitrary pass/fail filter can be set using the event_bool_fn type, which is elaborated on in Sect. 3.1.2.

Each procReweightor has a shared pointer to the process object it is meant to reweight (std::shared_ptr<REX::process> proc) and a vector of corresponding weightors (std::vector<weightor> reweight_functions) — function pointers to each corresponding reweighting function. These contain the central functionality for reweighting an individual process, performed by the member function evaluate(size_t amp):

```
void procReweightor::evaluate(size_t amp)
{...
auto wgts = this->reweight_functions
   [amp](*this->process)
...}
```

with some additional surrounding checks irrelevant for this description. Additionally, unless explicitly defined at the level of procReweightor, which process belongs to which procReweightor object is defined by the running reweightor and will be ignored here; for now, we assume there is a clear one-to-one relationship between processes and procReweightors.

At launch, the member function void initialise() should be called. After checking that the procReweightor has a process, initialise() checks whether the procReweightor has a member function pointer normaliser(process&); if it does not, it will set its normaliser to be the first available weightor, and if no weightors are available it will only return zero-valued weights. This is particularly important for hash misses, as elaborated on below, but for now it should be seen as a safety fallback if a procReweightor is initialised improperly.

Once initialise() has ensured that the procReweightor has a normaliser and a process, it will run

```
auto norm = this->normaliser(this->proc);
std::transform(norm->begin(),
   norm->end(), norm->begin(),
   [](double val){
   return (val == 0.0) ?
   0.0 : 1.0 / val; });
this->normalisation =
   *(REX::vec_elem_multi<double>
   (*norm, this->proc->weight());
```

i.e. for e.g. parameter reweighting it will set its normalisation factor to be $w/|\mathcal{M}|^2$, leaving all new weights to be normalised by a multiplication with the corresponding normalisation factor. Then, with this in mind, once the

[13] This can be overcome with more specific parton sorting in individual events, using details from Sects. 2.1.3 and 3.1.2. However, the default sorting routines provided by Rex only consider particle status (whether it is incoming, outgoing, or internal) and thus cannot treat anything more complex than differentiating between initial- and final-state particles.

[14] Due to the intricate ownership tree for most types provided by Rex it is plausible for objects to unintentionally fall out of scope — this is why shared pointers are so extensively used in the suite, aside from the contexts where it is important for objects to have shared ownership (such as processes and lhe).

procReweightor object gets the go-ahead to store a given reweighting iteration it will run

```
for(auto &wgts : this->backlog{
    this->proc->append_wgts(
    *(REX::vec_elem_multi<double>
    wgts, this->normalisation));}
```

where backlog is a member vector storing new weights between the call to evaluate() and the end of the current reweighting iteration. procReweightor members can be set using the self-returning setters

```
procReweightor &set_event_checker
   (REX::eventBelongs checker);
procReweightor &set_normaliser
   (weightor normaliser);
procReweightor &set_reweight_functions
   (weightor rwgt);
procReweightor &set_reweight_functions
   (std::vector<weightor> rwgts);
procReweightor &add_reweight_function
   (weightor rwgt);
procReweightor &set_process
   (std::shared_ptr<REX::process> pr);
```

although we reiterate that the process is generally intended to be set at the level of the reweightor type rather than the procReweightor.

As mentioned above, the reweightor type inherits from the REX::lhe type and adds additional functionality to sort owned events using owned procReweightors; run the process-specific reweighting functions; iterate over global states; appending resulting weights to the corresponding events; and calculating the resulting reweighted cross section σ' and corresponding cross section error $\Delta\sigma'$. Aside from inherited constructors and direct constructors from the lhe type, reweightor has the additional constructors

```
reweightor(lhe &&mother,
   std::vector<procReweightor> rws);
reweightor(lhe &&mother,
   std::vector<procReweightor> rws,
   std::vector<iterator> iters);
```

and corresponding constructors with std::shared_ptr<procReweightor> replacing procReweightor (which we reiterate are to be preferred), as well as corresponding copy constructors for the lhe object. In addition to the reweighting iterators shown in the constructors above, the reweightor type has two additional free iterators initialise and finalise intended to set and reset the global state to what it should be for and after the full reweighting procedure, respectively. As for most other Rex and teaRex types, reweightor has self-returning setters for most of its members:

```
reweightor &set_reweightors(
   std::vector<procReweightor> rws);
reweightor &add_reweightor(
   procReweightor &rw);
reweightor &set_initialise(
   iterator init);
reweightor &set_finalise(
   iterator fin);
reweightor &set_iterators(const
   std::vector<iterator> &iters);
reweightor &add_iterator(
   const iterator &iter);
```

and corresponding setters using std::shared_ptr<procReweightor>. One additional member that can be set as above are tags for individual reweighting runs, stored in the std::vector<std::string> launch_names which also has a setter and an element adder as above.

Internal details on the reweightor types are given in section 3.1.2, but the general algorithm which is run by calling the member function void run() is:

1. Call initialise()
2. Construct an event_hash_fn from procReweightors' owned eventBelongs
3. Sort events with the constructed hash
   - If there are unsorted events, add a procReweightor which returns weights zero
4. Return the resulting processes to the corresponding procReweightors which run their member normaliser on their process
5. Set up the threadPool of workers
6. For each owned iterator:
   (a) Run the iterator
   (b) For each procReweightor, submit a job to the threadPool
   (c) Once a job is launched; run each reweighting weightor owned by that procReweightor
   (d) Wait for all jobs to finish
   (e) Append returned weights to their processes
7. Transpose weights from processes to events
8. Call finalise()
9. If there are launch_names, add them to the common list of weight tags
10. Calculate reweighted cross sections and run error propagation for them

While this list is long, it is simple; in particular, most of the details are regarding the setup and wind down, while the individual reweighting iterations are simple. Note that reweighted cross sections and cross section errors are calcu-

lated despite not being stored as part of the LHE standard — using the default Rex writer they will not be written to disk, but they can be accessed in-software through the corresponding reweightor members

```
std::vector<double> rwgt_xSec;
std::vector<double> rwft_xErr;
```

which are given more detail in Sect. 3.1.2. Again, as a reminder, once a reweightor object has been set up using the constructors or setters above, the full reweighting procedure with all steps described above are performed by just calling the member function run(). The reweighted LHE file can then be written to disk using the default LHE writer or a custom writer as shown in Sect. 2.1.

3.1.2 Generic reweighting

While Sect. 3.1.1 describes most of the functionality and practical details of teaRex, there are some more involved possible uses, as well as some internal details, that may be of interest for more complicated implementations. These concern generic event hashing, details on the multithreading helper threadPool, the mathematical details of reweighted cross sections, and the possible implementation of single-event reweighting using the object-oriented event type rather than the SoA process type.

First, the procReweightor type has overloaded constructors and setters for the event_bool_fn type rather than the eventBelongs type. These allow for a generic way of defining which events belong to which reweightor in a less restricted format than that provided by the eventBelongs type (although obviously requiring more end-user programming), i.e.

```
procReweightor(weightor rwgt_fn,
   REX::event_bool_fn selector);
procReweightor(std::vector<weightor> rws,
   REX::event_bool_fn selector);
procReweightor(std::vector<weightor> rws,
   REX::event_bool_fn selector,
   weightor normaliser);
procReweightor &set_event_checker
   (REX::event_bool_fn selector);
```

However, this excludes the automatic parton indexing applied when using the eventBelongs type and consequently means the parton ordering in the transposed processes will be identical to that stored in the original event unless new indices are explicitly set in the selector.

Let us now turn to the threadPool type, used in teaRex to schedule and launch individual procReweightors across separate CPU threads. The type itself is a minimal wrapper for the std::vector<std::thread> member workers_, with size defined at construction,

```
threadPool::threadPool(unsigned int t){
   workers_.reserve(t);
   for (unsigned i = 0; i < t; ++i){
     workers_.emplace_back(
     ...
     );}
}
```

where the omitted section is just a lambda function for grabbing tasks from the memberstd::queue <std::function<void()>> q_ and error handling. To set up jobs, assuming the given tasks are stored in a vector of std::function<void()> jobs, is as simple as

```
std::vector<std::function<void()>
   jobs = {...};
threadPool pool(t);
pool.begin_batch();
for(auto job : jobs){
   pool.enqueue(job);
}
pool.wait_batch();
```

and the program will then wait until the batch is finished before continuing. By default, the pool used by the reweightor type is constructed with the number of threads available in the current context as provided by std::thread::hardware_concurrency(), but can be set explicitly using the reweightor member unsigned int pool_threads.

Next, we turn to the mathematical details of reweighted cross sections and error propagation. As mentioned, these are not part of the LHE standard itself (as they can be calculated from the cross section and individual event weights), but teaRex nevertheless provides functionality to evaluate them directly through the reweighting interface. Reweighted cross sections are given as

$$\sigma' = C \sum_i w'_i, \tag{6}$$

with C the same normalisation as the original event sample. This trivially extends to arbitrary observables as long as said observables are independent of the form of the hard scattering process, although teaRex currently only treats cross sections. Error propagation is performed assuming Gaussian behaviour as [22]

$$\Delta\sigma' = \Delta\sigma \cdot \left(\frac{1}{N}\sum_{i=1}^{N}\frac{w'_i}{w_i}\right) + \sigma \cdot \mathrm{std}(w'), \tag{7}$$

where by std we refer to the standard deviation of a variable. If error propagation for any reweighted cross section fails — by e.g. returning infinity, NaN, or non-positive —

a warning is raised without throwing an error and the error is estimated as the original error $\Delta\sigma$ multiplied by the ratio of the reweighted and original cross sections (conservatively always taking whichever ratio is greater than one).

One final point of consideration is the possible implementation of event-by-event reweighting using the OO `event` data format. While `teaRex` does not natively support this directly, to motivate the usage of the SoA `process` format for HEP software, it is possible to implement it indirectly by noting that the definition of the `weightor` type,

```
using weightor = std::function<
   std::shared_ptr<std::vector<double>>
   (process&)>;
```

does not enforce the usage of any specific members of the `process` type, only the usage of the `process` type itself. As mentioned in Sect. 2, the `process` objects owned by `lhe` objects have shared access to their corresponding `event`s through vectors of shared pointers to `event`s, i.e.

```
std::vector<std::shared_ptr<event>>
   lhe.processes[i]->events =
   lhe.sorted_events[i];
```

from which it immediately follows that event-by-event reweighting can be performed in `teaRex` by wrapping a loop over individual `event` reweighting calls in a `weightor`, e.g.

```
double foo(const REX::event &ev){...}
REX::tea::weightor fooWrap = [](
  std::shared_ptr<std::vector<double>> pr)
  {auto wgts = std::make_shared
      <std::vector<double>>({});
   for(auto e : pr.events){
      wgts->push_back(foo(*e));}
   return wgts;
  };
```

and while not directly supported nor recommended, this is a possible use of `teaRex`.

3.1.3 SLHA parameter reweighting

As part of `teaRex` an explicit reweighting implementation is provided, used as the basis for SLHA parameter reweighting in MadtRex. A detailed description of it is provided below, illustrating how to implement the `teaRex` library.

The SLHA parameter reweighting is defined using just two new types besides those already implemented in `Rex` and `teaRex`: `rwgt_slha`, which generates SLHA parameter cards for reweighted parameters and writes them to disk; and `param_rwgt`, a small `reweightor` child type with ownership of a `rwgt_slha` object to generate its `iterator`s. How parameter reweighting is implemented in `teaRex` works is algorithmically simple: There exists some externally provided scattering amplitude functions in the `weightor` format, and these `weightor`s read model parameters from a parameter card on disk. The `rwgt_slha` object is provided with a reweighting card with the format

```
launch rwgt_name=run1
set BLOCK_NAME PARAM_ID VALUE
set BLOCK_NAME PARAM_ID VALUE
launch rwgt_name=run2
set BLOCK_NAME PARAM_ID VALUE...
```

as well as an original SLHA parameter card to modify the parameter values in. Inheriting from the `REX::slha` type, these commands are easily translated to `iterator`s which overwrite the parameter card with one containing the new parameters, as well as an `initialise` function copying the original card to a safe location and a `finalise` function moving the original card back into position. Aside from surrounding safety and sanity checks, this is implemented as

```
bool write_rwgt_card(size_t idx){
for(auto [key,val] : cards[idx].blocks){
   for(auto [p_id, p_val] : val.params){
      original.add_param(key, p_id,
         this->get(key, p_id));
      this->set(key, p_id, p_val);}
   }
this->write(std::ofstream(card_path));
}
```

with some simplifications made for easier reading, noting that the member `original` is an owned `REX::slha` object storing the original values of modified parameters to ensure the original value of reweighted parameters are reset before reweighting new ones. In short, `write_rwgt_card` writes SLHA parameter cards identical to the original card save for the parameters modified for that particular reweighting iteration (as given by the launch commands mentioned above). The reweighting `iterator`s are provided by

```
std::vector<iterator> card_writers(){
std::vector<iterator> writers;
for(size_t i=0; i < cards.size(); ++i){
   writers.push_back([this,i]){
      return write_rwgt_card(i);
   };}
return writers;
}
```

and from this the entire foundation for a `reweightor` is presented; `teaRex` provides `process`-specific reweightors using `weightor`s, but in this case a global variable (i.e. model parameters) is modified using `iterator`s before running the same `weightor` routines which are dependent on the global state set by the `iterator`s. In fact, the few additional members of the `param_rwgt` type besides those inherited from `reweightor` are only there to interface

directly with the rwgt_slha type to minimise necessary interfacing in implementations: param_rwgt has a single unique member function, which passes a reweighting card in the format above to its rwgt_slha member and then initiates its own initialise, finalise, and iterators to those provided by the rwgt_slha:

```
void read_slha_rwgt(std::istream &slha,
   std::istream &rwgt){
   card_iter = rwgt_slha::create
      (slha, rwgt);
   initialise = [&](){
   return card_iter.move_param_card();};
   finalise = [&](){
   return card_iter.remove_param_card();
   };
   iterators = card_iter.
      get_card_writers();
...}
```

with some additional lines afterwards passing information about the reweighting onto the lhe context for writing. Fundamentally, though, param_rwgt is just a reweightor with iterators provided by the commands given in a reweighting card in the SLHA format; besides that, all it needs are weightors alongside corresponding event sorters.

3.2 Use case illustrations

As mentioned above, teaRex is a relatively small library when compared to Rex, and we hope its usage to be clear from the descriptions above. For a slightly more detailed illustration, see algorithms 5 and 6 which detail the implementation in MadtRex. However, as an illustrative example let us outline an implementation of pdf reweighting using teaRex; an example of such a program is shown in algorithm 3, and we will now continue to go through some details of an implementation of this.

The first thing to consider is the subprocess definition; for this illustrative example, we consider only initial-state partons (i.e. particles with status=-1) of which we assume there will always be two, and we further assume that these will be either massless quarks (defined as quarks belonging to the first two generations) or gluons. The three resulting subprocesses are ones with either two quarks, two gluons, or one of each[15]. As we define our procReweightors and consequently the event_hash_fn used to sort the lhe object using the eventBelongs type, the transposed process objects will be filtered to only these initial-state partons which will furthermore always have the ordering specified by the events used in algorithm 3.

Once the eventBelongs objects are defined, corresponding REX::tea::weightors need to be loaded. The example in algorithm 3 only supplies a single weightor, but these could equally well be std::vectors of weightors with entries for each pdf set. In that case, the procReweightor member normaliser should also be set to define the original weight with which reweighting is performed with respect to. Alternatively, the format shown in algorithm 3 can be used alongside a global wrapper and iterator which is cycled through using the reweightor member iterators, as shown in algorithm 4, where in this minimal implementation it is clear that precautions need to be taken with regard to the sizes of the vectors of pdf sets, and we note that reweightor calls iterators *before* running weightors, meaning in algorithm 4 the first element of the vectors of functions should be the original pdf set. Implementations using global function wrappers and iterators are likely to be slower than ones with vectors of weightors due to the required sync between reweightor and procReweightors, but may be simpler or necessary depending on the specific structure of the reweighting functions. For pdf reweighting, specifically, this should not be an issue, but for e.g. MadtRex where scattering amplitude routines read physics parameters from disk keeping iterations in sync is imperative.

Once the procReweightors have been defined and the lhe object initialised, the reweightor can be constructed directly using the explicit reweightor constructors, and all the intricacies of sorting and transposing events, running reweighting iterations and normalising, and appending the new weights to the lhe are done automatically by calling the reweightor::run() function.

[15] The implicit fourth subprocess consisting of any events in our sample that fail these conditions is assumed to be a mismatch for the reweighting procedure and thus is given zero weights for each weight appended to the sample. In a procedure like pdf reweighting this may not be the intention, and one could add an explicit "all-encompassing" fourth subprocess using the pre-defined eventBelongs returned from REX::all_events_belong() which, as the name suggests, just returns true for all events. Combining this with a trivial REX::tea::weightor that only returns e.g. std::vector<double> ones(process.events.size(), 1.0) and placing it as the very last procReweightor ensures any events that have at least one different initial-state parton will maintain their original model weight. The procedure to treat events with only one initial-state quark would necessitate either an eventBelongs object with all possible additional initial-state partons *or* a custom event_bool_fn, neither of which would be particularly difficult to implement.

Algorithm 3 General outline for a program to run pdf reweighting using `Rex`, including explicit definitions of sorting operators to split the sample into events with two initial-state quarks, two initial-state gluons, or one of each. The actual pdf sets have been omitted, but would be provided through the `tea::weightor` objects listed in the code, and could either be implemented as global functions with `tea::iterator`s changing them globally between iterations or instead as vectors of `tea::weightor`s with one element per pdf set. The only assumption here is that the order of the initial-state particles is unimportant, i.e. that the same pdf set will be used for both beams: using separate ones per beam would necessitate a custom `event_bool_fn` (or at least a custom `event_comp_fn`), as `Rex` does not have support for ordering partons explicitly based on momenta.

```
using namespace REX;
std::vector<event> gg, gq, qq;
std::vector<int> qs =
  {1,-1,2,-2,3,-3,4,-4};

for(size_t q1 = 0; q1 < qs.size(); ++q1){
 for(size_t q2 = q1; q2 < qs.size(); ++q2)
 {event ev_qq(2).set_status({-1,-1})
    .set_pdg({qs[q1],qs[q2]});
  qq.push_back(ev); }
 event ev_gq(2).set_status({-1,-1})
   .set_pdg({21,qs[q1]});
 gq.push_back(ev_gq);
}

gg.push_back(event(2).set_status({-1,-1})
  .set_pdg({21,21}));

auto comp = eventComparatorConfig()
  .set_pdg(true).set_mass(true)
  .set_status_filter({-1})
  .make_comparator();

eventBelongs quark(qq, comp);
eventBelongs mixed(gq, comp);
eventBelongs gluon(gg, comp);

std::vector<tea::procReweightor> rwgtrs;

tea::weightor quark_pdf_fn = ...
tea::weightor mixed_pdf_fn = ...
tea::weightor gluon_pdf_fn = ...

rwgtrs.push_back(tea::procReweightor(
   quark_pdf_fn, quark);
rwgtrs.push_back(tea::procReweightor(
   mixed_pdf_fn, mixed);
rwgtrs.push_back(tea::procReweightor(
   gluon_pdf_fn, gluon);

lhe file_to_reweight = ...

tea:reweightor rwgt_runner(
   file_to_reweight, rwgtrs);

rwgt_runner.run();
```

Algorithm 4 Minimal illustration of a pdf reweighting helper class to modify pdf sets globally rather than supplying a vector of function pointers to individual pdf sets.

```
class my_pdfs{
private:
   std::vector<REX::tea::weightor>
     qq_pdfs, gq_pdfs, gg_pdfs;
   size_t curr_pdf;
   bool increment()
   { ++curr_pdf; return true; }
public:
   std::shared_ptr<std::vector<double>>
    qq_pdf(REX::process& p)
    { return qq_pdfs[curr_pdf](p); }
// And similarly for gq_pdf(s), gg_pdf(s)
...
   std::vector<REX::tea::iterator>
    get_iterators()
    {
      std::vector<REX::tea::iterator>
       iters(qq_pdfs.size(), &increment);
      return iters;
    }
}
```

4 MG5aMC `teaRex` reweighting executables

The MadGraph5_aMC@NLO TeaRex Reweighting Extension — MadtRex (pronounced *metrics* or *matrix*) — is an extension to the `CUDACPP` plugin [12–17] for MadGraph5_aMC@NLO (MG5aMC) [21] repurposing the scattering amplitude routines written for data-parallel event generation as a basis for data-parallel event reweighting using the `teaRex` library. Specifically, MadtRex enables model parameter reweighting with an alternate backend for the MG5aMC reweighting module [22] built with `teaRex` — specifically the SLHA backend presented in section 3.1.3 — and compiled libraries of the process-specific scattering amplitudes generated by `CUDACPP`.

Below, we detail the usage of and speed-up provided by MadtRex when compared to MG5aMC reweighting. Section 4.1 provides an in-depth manual for using MadtRex in the context of the `CUDACPP` plugin, including installation, usage, and for the interested reader a description of the underlying implementation. In section 4.2 we then present runtime comparisons between MadtRex and the default MG5aMC module, including some discussion on the different sources of speed-up of which there are several beyond the hardware acceleration provided by `CUDACPP` scattering amplitudes.

4.1 Manual

While MadtRex uses the exact same interface as (generic) reweighting in MG5aMC, as the Python-side reweighting module itself is inherited from the MG5aMC reweighting module (with modified code generation and execution launchers), there are some details worth mentioning. This manual describes the installation of MadtRex in section 4.1.1 and how to use it for parameter reweighting in Sect. 4.1.2. For details on the implementation of the reweighting executable program, see section 4.1.3.

4.1.1 Installation

MadtRex has been integrated into the `CUDACPP` main repository alongside copies of the 1.0.0 releases of `Rex` and `teaRex`, and will be included in all upcoming releases corresponding to MG5aMC v3.6.4 and onward. For more extensive details on installing `CUDACPP` refer to [17], but we note that as of MG5aMC version 3.6.0 `CUDACPP` can be installed directly through the MG5aMC CLI using the command

```
MG5_aMC> install cudacpp
```

with the optional additional argument `–cudacpp_tarball=URL` with `URL` the URL of a specific `CUDACPP` release provided as a tarball.

For the time being, updates to `Rex`/`teaRex` are not automatically propagated to the `CUDACPP` repository; furthermore, manual updates of the copies provided with `CUDACPP` are only anticipated for major `Rex` or `teaRex` releases and even then likely only if the updates are expected to improve MadtRex performance explicitly. However, alternate versions of these libraries can of course be manually installed by the end-user.

Changing `Rex` and `teaRex` releases is as simple as overwriting the existing ones and recompiling. The files in question are `Rex.h`, `Rex.cc`, `teaRex.h`, and `teaRex.cc`, all stored in the directory

```
/mg5amcnlo/PLUGIN/CUDACPP_OUTPUT/MadtRex/
```

and can be compiled using the minimal command `make -f rex.mk`.

4.1.2 Usage

MadtRex inherits the interface of the upstream MG5aMC reweighting module, although it has some restrictions that the latter lacks. Unlike MG5aMC, which supports reweighting at both leading and next-to-leading order, MadtRex is restricted to leading order reweighting; furthermore, the default reweighting mode in MG5aMC is helicity-exclusive (i.e. the reweighted event is only evaluated at the same helicity configuration as in the original model), whereas MadtRex is limited to helicity-summed reweighting in both the original and reweighted model due to the lack of helicity-specific scattering amplitudes supported by `CUDACPP`[16]. Aside from these restrictions, MadtRex supports reweighting to and from any leading order (tree-level) model supported by the `CUDACPP` plugin.

Once installed, MadtRex reweighting can be enabled by setting the MG5aMC `reweight` flag to `madtrex` at program launch, i.e. once in the MG5aMC command line interface running the commands

```
generate PROCESS
output DIRECTORY
launch
reweight=madtrex
...
```

with the final written line telling MG5aMC to utilise the `madtrex` reweighting module rather than the default reweighting module shipped with MG5aMC itself. The `reweight_card.dat` can then be modified to include values for the desired hardware backend, floating point precision, and number of CPU threads to launch from the host executable, using the commands

[16] There is ongoing work in implementing single-helicity scattering amplitude calls in `CUDACPP`; propagating this to MadtRex in a scaleable way (i.e. without treating each external helicity configuration as a separate subprocess) will take additional work, but is considered for future developments.

```
change backend BACKEND
change fptype FPTYPE
change nb_thread NB_THREAD
```

which must be appended before the first `launch` command. Any strictly positive integer is allowed for `nb_thread`, while supported options for the compile-time arguments are

- `backend`: `cppauto`, `cppnone`, `cppsse4`, `cppavx2`, `cpp512y`, `cpp512z`, `cuda`, `hip`. To reweight on a SIMT GPU, set the backend to the corresponding framework (i.e. CUDA for Nvidia GPUs or HIP for AMD GPUs), otherwise we recommend using the default `cppauto` which automatically detects the best SIMD instructions supported by the machine.
- `fptype`: `m` (mixed precision), `d` (FP64), and `f` (FP32). Default is `m`, which computes scattering amplitudes in FP64 and colour algebra in FP32. We recommend avoiding `f` due to the risk of catastrophic cancellations between Feynman diagrams, but leave the option available.

For further details on these options, consult CUDACPP documentation [17].

Once the reweighting card has been set, keep running the event generation as normal and MadtRex will take care of the rest. If no compiled versions of `Rex` or `teaRex` are detected, they will be compiled before code generation for the MadtRex executable. This may take some time but only needs to be done once, after which the library can be linked against for all future MadtRex calls on the same machine.

In the backend, the MadtRex reweighting module inherits from the upstream MG5aMC reweighting modules with modified code generation primarily inherited from the CUDACPP code generation types alongside the additional infrastructure needed to compile individual parton configurations into libraries loaded into a single MadtRex executable, with the code compilation step modified to account for the new structure. The actual reweighting execution, which in the MG5aMC reweighting module is done entirely within the Python module itself, has been scrapped and overridden with two small functions: the first parses the more generic MG5aMC reweighting card and rewrites it to the stricter format assumed by MadtRex executables; and the second launches the compiled MadtRex executable.

4.1.3 Executable details

MadtRex executables are relatively simple programs, in the sense that all functionality is provided by `Rex`, `teaRex`, and CUDACPP-generated scattering amplitude functions. The reweighting iterations are provided by the `REX::tea::param_rwgt` type detailed in Sect. 3.1.3, and we will forego repeating the details here, but do note that the Python CLI driver will translate the `reweight_card.dat` to the more specific standard required by `teaRex`, meaning MadtRex has support for user-friendly MG5aMC commands such as parameter names (e.g. `aEW` for α_{EW}) or scans (i.e. automatic reweighting over several parameter values, e.g. `set aEW scan:[100,150,200]`).

With these surrounding details already provided, we turn to the implementation of CUDACPP scattering amplitudes as distinct libraries to be included within a single MadtRex executable. For details on CUDACPP itself and the details of how scattering amplitudes are interfaced with the MadEvent event generator, see [17]. The only necessary points to mention here are that CUDACPP generates data-parallel scattering amplitude evaluation routines with a minimal bridge API to allow other programs to access these routines by providing the relevant process data in a column-major SoA format.

Fundamentally, MadtRex executables have three parts stored across three separate files: the generic `rwgt_instance`, which defines the functionality connecting the CUDACPP API and the executable itself; the process-specific `rwgt_runner`, providing an `event_bool_fn` to specify `procReweightor`s and a wrapper for the specific scattering amplitude related to that `process`; and the executable `rwgt_driver`, sorting out the different `rwgt_runners` and constructing a `reweightor` from them and the LHE file to be reweighted. The latter two are generated by MadtRex for a given process to be reweighted — `rwgt_driver` less so than `rwgt_runner` — while the first one simply defines the interface between the latter two. Since the structure of the `rwgt_instance` files is primarily to handle the details of the CUDACPP API, we forego details.

Turning first to the `rwgt_runner.cc` file, it holds exactly four functions, with two of them being functionality wrappers. These are identically named across `rwgt_runners`, but each one is wrapped in a namespace defining its particular subprocess. The first function, `get_comp`, creates an `eventBelongs` object corresponding to the specific scattering amplitude code for this particular subprocess. Since this will depend entirely on which amplitudes the given subprocess is to evaluate, this function is generated independently for each generated subprocess. An example of this function for reweighting the process $l^+l^- \to l^+l^-$ is provided in algorithm 5, where an `eventBelongs` object testing whether a given `event` corresponds to either of the hard scattering processes $e^+e^- \to e^+e^-$ or $\mu^+\mu^- \to \mu^+\mu^-$ is constructed.

The next function in `rwgt_runner.cc` is `amp`, which is a call to the CUDACPP API wrapped in the process-specific namespace to avoid symbol overlap between subprocesses; specifically for parameter reweighting, the only `process` information passed on to the amplitude routine are particle

Algorithm 5 The `get_comp()` function provided in one of the `rwgt_runner.cc` files for MadtRex reweighting of the standard model LO process $l^+l^- \to l^+l^-$. Note that both the `stats` and `pdgs` vectors consist of two elements, corresponding to the two sets of external legs this particular subprocess evaluates, in this case when the initial- and final-state particles are identical.

```
std::shared_ptr<REX::eventBelongs> get_comp()
{ static std::vector<std::vector<short int>> stats = {{-1,-1,1,1},{-1,-1,1,1}};
  static std::vector<std::vector<long int>> pdgs = {{-11,11,-11,11},{-13,13,-13,13}};
  static std::vector<std::shared_ptr<REX::event>> loc_evs;
  for (size_t i = 0; i < stats.size(); ++i)
  {  auto ev = std::make_shared<REX::event>(pdgs[i].size());
     ev->set_status(stats[i]);
     ev->set_pdg(pdgs[i]);
     loc_evs.push_back(ev); }
  return std::make_shared<REX::eventBelongs>(loc_evs, REX::external_legs_comparator); }
```

Algorithm 6 The `main` function for the MadtRex executable `rwgt_driver.cc`, argument handling omitted. All functionality shown is either directly from `Rex`, `teaRex`, or the scattering amplitude wrappers in the `rwgt_runners`.

```
int main(int argc, char **argv){
   std::cout << "Starting MadtRex driver...\n"
   ...
   static std::vector<std::shared_ptr<REX::tea::procReweightor>> rwgtRun =
      {P1_Sigma_sm_epem_epem::make_reweightor(),
       P1_Sigma_sm_epem_mupmum::make_reweightor(),
       P1_Sigma_sm_epmum_epmum::make_reweightor()};
   auto rwgt_runner = REX::tea::param_rwgt(REX::load_lhef(lheFilePath), rwgtRun);
   rwgt_runner.read_slha_rwgt(slhaPath, rwgtCardPath);
   rwgt_runner.pool_threads = nb_threads;
   rwgt_runner.run();
   std::cout << "\nReweighting procedure finished.\n";
   std::ofstream lhe_out(outputPath);
   if (!lhe_out)
      throw std::runtime_error("Failed to open output LHE file for writing.");
   rwgt_runner.print(lhe_out, true);
   std::cout << "Reweighted LHE file written to " << outputPath << ".\n";
   ...
   return 0;
}
```

momenta, accessed through the call `process::pUP().flat_vector()`. The final two functions are both wrappers — one, `bridgeConstr`, for creating a `rwgt_instance` object handling the intricacies of going from a `process` to the expected arguments for CUDACPP; and the other, `make_reweightor`, creating a `procReweightor` from the bridge just mentioned as well as the `get_comp()` function illustrated in algorithm 5. The only one of these functions called directly from `rwgt_driver` is `make_reweightor` — all other intricacies are handled by `teaRex`[17].

Aside from argument and error handling, `rwgt_driver.cc` is also a very simple file. At code generation, the corresponding `rwgt_runner.h` files are added to the list of included header files, and one additional line is written constructing a vector of `procReweightor`s consisting of the return value from each `make_reweightor` function. Returning to the process $l^+l^- \to l^+l^-$ and omitting argument handling, the full MadtRex executable `main` function is shown in algorithm 6. All the executable needs to do is define the `procReweightor`s, given by an `eventBelongs` object and a `weightor` (since parameter reweighting using MadtRex assumes identical `weightor`s for normalisation and reweighting); define the `iterator`s — in this case, a set of functions overwriting the SLHA parameter card read by the `weightor`s — and call `reweightor::run()`. All details are sorted out by `teaRex`, illustrating the ease of use the library provides.

[17] Do note, however, that the `rwgt_runner.cc` files and all the surrounding scattering amplitude functionality it accesses have significant symbol overlap, as CUDACPP uses the same names across subprocesses and internally uses no process-specific namespaces. In MadtRex this is overcome using the linker flag `-Bsymbolic`, which binds references to global symbols to the definition within the shared library.

4.2 Runtime comparisons

As a simple illustration of the speed-up provided by MadtRex — with respect to the various data-parallel backends enabled by `CUDACPP`-provided scattering amplitudes as well as in comparison to the reweighting module provided by MG5aMC — we consider a realistic use case for tree-level parameter reweighting, in reweighting SM samples (generated at arbitrary order, although we here stick to leading order for simplicity) to BSM models, allowing for the study of BSM effects on observables by reweighting simulated samples to new models.

For this benchmark, we turn to SM effective field theory (SMEFT), where the SM is interpreted as an effective field theory of a higher-dimensional model and allows for generic parametrisation of BSM effects, assuming the higher-dimensional model abides by the same global symmetries as the SM [33]. Further details on the SMEFT are unimportant here; it is sufficient to note that the SMEFT includes a plenitude of free parameters but reduces to the SM at lower energies, making it an ideal target for simulation recycling using parameter reweighting.

We will consider 4-top production in the SMEFT, i.e. the process

$$p\,p \to t\,\bar{t}\,t\,\bar{t} + n \text{ jets}, \tag{8}$$

for $n \in \{0, 1\}$ and any massless QCD jets. Furthermore, we will consider both the generic process with p, j any massless QCD parton ("multi-channel") and the case where p, j are both set to be exclusively gluons such that there is only the computationally heaviest gluonic integration channel ("single-channel").[18] For these four processes, we first generate SM samples (at leading order for practicality) with between 10 and 10^7 events. Then, we reweight these samples to various different sets of Wilson coefficients for the top-related couplings in the "$U(3)_l \times U(3)_e$-symmetric" SMEFT, for which SMEFTsim provides the UFO model `SMEFTsim_topU3l` [23].

To determine the throughputs of the different implementations — MG5aMC reweighting and MadtRex with scalar instructions, SIMD instructions, and GPU offloading — we start at small sample sizes and few parameter sets reweighted to, and then increase both until a plateau is reached (considered the point at which the throughput is no longer consistently increasing with the number of reweighted events and reweighting iterations). This will of course vary for the different implementations: for MG5aMC the plateau was generally reached already for 100 events and 8 reweighting iterations, while for MadtRex it was typically necessary to run at least 36 iterations for 1 000, 10 000, and 100 000 events needed to reach the plateau for scalar instructions, AVX2 instructions, and GPU offloading using an Nvidia A100, respectively,[19]. These measurements are shown in Fig. 5, where `nb_thread=1` for all MadtRex executions.

Figure 5 provides extensive insights, but none particularly surprising. Since parameter reweighting is a computationally bound problem completely dominated by scattering amplitude evaluations (the exact part parallelised in `CUDACPP`-generated code) AVX2 instructions provide a speed-up of roughly a factor 4 compared to scalar instructions, i.e. the exact speed-up provided for 64-bit floating point (FP64) operations between scalar and AVX2 instructions assuming a constant CPU clock speed. This suggests perfect data parallelism, and that in the high complexity many-event limits, MadtRex throughput will scale one-to-one with SIMD register size. Additionally, for complicated processes like these, GPU offloading using a high-performance general-purpose GPU like the Nvidia A100 can provide further speed-up when compared to on-host SIMD parallelism; in Fig. 5, the representative single-channel processes gain a speed-up of $2-5$ times using GPU offloading when compared to on-CPU reweighting, which we imagine is limited by host-device communication latency and could be improved by e.g. rewriting amplitude evaluation routines to store model parameters internally rather than loading them from disk. By default MadtRex makes use of FP64, and peak throughputs can be estimated on a process-by-process basis by comparing the FP64 floating point operations per second across hardware, which at least for top-end Nvidia GPUs is stagnating at the level supplied by the generation following the A100 (i.e. the Nvidia H100 line) [34][20]. Further acceleration could be achieved by compiling MadtRex executables for lower precision at risk of significant floating point errors (see [17]).

More noteworthy is the sizeable speed-up when comparing scalar MadtRex with the native MG5aMC reweighting module: For all tested processes, MadtRex execution *without* any explicitly implemented data parallelism has a throughput $30-60$ times greater than MG5aMC. There are

[18] By single- and multi-channel, we refer to the number of external partonic configurations integrated over during event generation; for the multi-channel processes, all valid combination of gluons and massless quarks are included in the sample, whereas the single-channel samples only include the gluonic channel during both event generation and reweighting.

[19] Specifically, reweighted parameter sets were defined in terms of linear to octic power combinations of the considered Wilson coefficients. For MG5aMC no power beyond cubic (36 reweighting iterations) could be finished within a reasonable time frame, while for MadtRex with GPU offloading the octic power combinations (6435 reweighting iterations) would finish for samples of 100 000 events within a couple of hours.

[20] Note that peak FP64 performance for AMD GPUs is *not* stagnating and has outpaced Nvidia GPUs recently [35]. However, the corresponding performance has not been seen when running `CUDACPP` on AMD GPUs, which is currently being investigated.

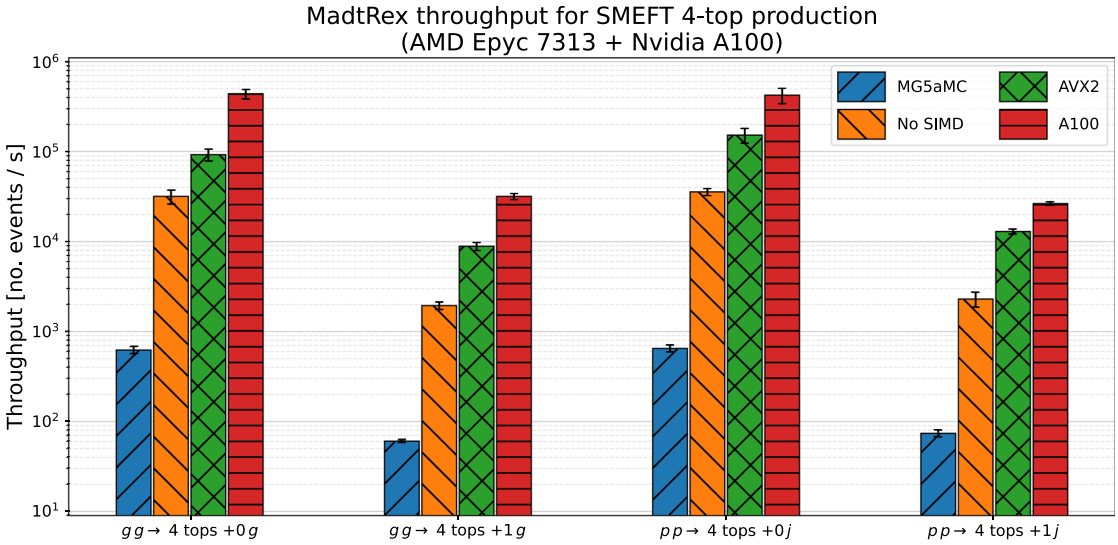

Fig. 5 Event throughput for MADTREX reweighting as well as the default MG5AMC reweighting module for comparison. Throughputs and standard deviations have been calculated based on mean runtimes for various event samples (ranging from 10 to 10^7 events) with various number of reweighted parameter sets (ranging from 8 to 6435 iterations). Although GPU offloading has a clear advantage over on-host SIMD parallelism, which in turn is faster than scalar instructions, MADTREX is consistently $\sim$ 40 times faster than MG5AMC reweighting even without explicitly implemented data parallelism

two reasons for this: (1) rather than sorting events online for each reweighting iteration (as MG5AMC does), MADTREX has a "one-and-done" upfront sorting algorithm; and (2) MADTREX runs a compiled reweighting executable rather than a Python driver calling Fortran functions through `f2py`.

Point (1) reduces the leading constant in the linear runtime growth by limiting it to only the number of events, i.e. the sorting runtimes grow as

$$t_{\text{MG5AMC}} = \mathcal{O}\left(\#\text{events} \times \#\text{iterations}\right), \tag{9}$$

$$t_{\text{MADTREX}} = \mathcal{O}(\#\text{events}), \tag{10}$$

minimising the runtime cost of reweighting to additional parameter sets.

When considering available computational power scaling, however, point (2) is more interesting: the structure of MG5AMC reweighting limits it to running sequentially single-threaded, and overcoming this would require explicit modifications to the reweighting module. With MADTREX, however, this is automatically provided through compiler optimisation. As mentioned, the tests shown in Fig. 5 were run with `nb_thread=1`, meaning no multithreading over subprocesses; however, compiler optimisation can still enable multithreading *within* a given subprocess. This has been directly observed: Running on-CPU MADTREX executables, child processes are consistently launched across 8 CPU cores without any multithreading across subprocesses,[21]. We can assume that a factor $\sim$ 8 of on-CPU MADTREX speed-up thus comes from in-subprocess multithreading provided directly from compiler optimisation, leaving the speed-up from not using an interpreted language at $\sim$ 5. Whether this can be scaled further will be considered for continued `teaRex` and MADTREX development.

However, this benefit does not apply for GPU offloading, as in-subprocess multithreading is already the target of the CUDA-compiled MADTREX executables. As evidence of this, consider the following: the single-channel gluonic processes are the most computationally heavy subprocess of the multi-channel processes. Consequently, for a computationally bound problem, the throughput for multi-channel processes should be equal to or greater than the single-channel ones, which is seen for all on-CPU reweighting implementations. On the other hand, the addition of multiple subprocesses, which are run sequentially, will impact a latency-dominated executable, and as Fig. 5 shows, the GPU execution for multi-channel processes is consistently slower than single-channel processes. With this in mind, for few-CPU MADTREX jobs there is little reason to try to optimise the `nb_thread` variable for on-CPU jobs, while it could be essential for making the best use of GPU offloading. Although we have not studied performance across many-

[21] Due to how explicit multithreading is set up in MADTREX the single-channel processes with only parton configuration cannot benefit from subprocess multithreading since these processes consist of a single subprocess in the MADTREX scheme.

CPU systems, the same considerations as for GPU offloading likely still apply as latency may become a predominant bottleneck for the event-level multithreading compiler optimisation appears to implement. Do note that MadtRex does not currently support multi-GPU setups and will only target a single GPU on such a machine; should multi-GPU reweighting be requested by users, we will consider its implementation.

Regardless of the specifics, it is clear that MadtRex provides both immediate speed-up when compared to MG5aMC reweighting, and great potential for better scalability across larger and distributed systems. Although MadtRex currently only provides functionality for tree-level LO parameter reweighting, this alone enables the reuse of already simulated SM samples for the study of BSM models such as the SMEFT used here. In the long term, further developments in `CUDACPP` functionality towards NLO event generation could enable NLO parameter reweighting in MadtRex with minimal development necessary from the MadtRex side.

5 Conclusions

The three codes presented in this paper — `Rex`, `teaRex`, and MadtRex — provide an accessible entry point for HEP software handling parton-level hard scattering events. `Rex` provides a physics-oriented interface for LHE file format events while providing tools for the simple implementation of I/O for further LHE-like file formats, and furthermore enables trivial transposition between human-readable OO data formats and SoA formats designed for data-parallel hardware acceleration. These data formats are used as a basis for completely generic event reweighting in `teaRex`, which provides a basis structure for reweighting events to arbitrary conditions, necessitating users to only provide the corresponding reweighting functions. Using `Rex` for event data handling and `teaRex` as a basis, the MadtRex reweighting module enables data-parallel model parameter reweighting within MG5aMC using the `CUDACPP` plugin as a basis for scattering amplitude evaluations, and only using the `Rex` sorting algorithm and compiler optimisation consistently achieves reweighting throughputs $30-60$ times greater than the default MG5aMC reweighting module for computationally complex processes; using on-CPU SIMD instructions increases this throughput further by the expected maximal gain for AVX2 instructions, and GPU offloading can push the total acceleration up to a factor $300-700$ depending on the process. `Rex` and `teaRex` are currently available on GitHub at the URL https://github.com/zeniheisser/Rex with versions 1.0.0 archived [36] on Zenodo [37]. MadtRex is included as part of the `CUDACPP` plugin alongside versions 1.0.0 of `Rex` and `teaRex` as of October 22nd 2025.

Going forward, all three codes have great potential for further development. Starting with `Rex`, ensuring the data access interface is as simple as possible is and will always be the main concern, although what exactly this entails remains to be seen based on user feedback. Some possibilities, though, include bindings for more commonly used programming languages — particularly Python — allowing its usage across a far wider range of software than just compiled C++ programs. Additional data access functionality is also simple to implement and will be considered upon request. One particular point of further consideration is whether to attempt to optimise `Rex` for memory consumption; at present, `Rex` data takes up roughly the same size in memory as the LHE plaintext format does on disk, which may or may not be a limiting factor depending on the sizes of samples used for practical applications. Alternatively, should I/O speed be a major consideration for users we may write an interface for the LHEH5 format to ship alongside the current standard LHE interface. Generally, we are open to feature requests with higher priority assigned to users who already have implemented or are actively working on `Rex` implementations.

While minimal in size, `teaRex` has already proven to be extremely potent at its intended purpose of enabling simple implementation of (data-parallel) parton-level event reweighting. Furthermore, being an extension to `Rex`, `teaRex` will benefit directly from any additional development in `Rex`. Considering specifically `teaRex` though, there are some potential avenues of further development: first, `teaRex` only provides explicit multithreading support across separately defined subprocesses of a given event sample; this does not necessarily make the best use of available compute, and applying further subprocess splitting where subprocesses with many events are divided into additional separate execution tracks may make better use of on-CPU multithreading. While we have not identified further optimisations in the `teaRex` structure itself, we are open to user suggestions; however, we expect ease of use to be a more interesting concern, just as for `Rex`. Like we suggested for `Rex`, we expect Python bindings (or similar) to be a valuable future development to allow for the automatic data-parallel reweighting of generic reweighting using *any* input functions without needing to implement an executable program.

Further MadtRex development is unlikely to focus on optimisation; `CUDACPP` scattering amplitudes are already perfectly parallel and by virtue of only calling the scattering amplitudes themselves MadtRex makes perfect use of them. Of course, MadtRex will benefit from any optimisation in `CUDACPP` amplitudes, but more importantly it can gain extended functionality from added features to `CUDACPP`. Of greater interest for the average user, though, may be that `CUDACPP` is planned to become the default (leading order) code generator for MadGraph as part of the upcoming MadGraph7 project, and that there is active discussion about making MadtRex the default reweighting module as part of that development. Thus, further MadtRex develop-

ment is likely to be focused on integrating it further into the MG5AMC architecture and extending its functionality to treat other reweighting use cases within the MG5AMC suite, such as NLO parameter reweighting or pdf reweighting.

Overall, `Rex` and `teaRex` provide an efficient parton-level event interface for HEP software, enabling the trivial transposition between OO and SoA data formats as well as the generic reweighting of events using completely generic functions for whatever parameters are being reweighted. Implementing this as well as proving its applicability, the MADTREX reweighting module for MG5AMC can provide a $30-70$ times throughput increase for computationally heavy processes on the exact same CPU without any explicitly implemented data parallelism, while AVX2 instructions increase this to a factor $150-300$ speed-up and GPU offloading using an Nvidia A100 GPU can push it as far as a factor 700 throughput increase. Further developments in all codes are likely to primarily consider functionality and ease of access as well as potential integration into existing codebases in order to provide these benefits for as large a fraction of the community as possible.

Acknowledgements We extend our gratitude to Olivier Mattelaer for discussions regarding the interfacing used in `Rex` and `teaRex`, and in particular for his assistance in enabling integration between MADGRAPH5_AMC@NLO and MADTREX; additional thanks are extended to Andrea Valassi for his assistance in the interfacing between `teaRex`, MADTREX, and the `CUDACPP` plugin; and to Stephan Hageböck for discussion and recommendations regarding efficient data interfacing between different data formats with respect to LHE parsing and storage within `Rex`. Additionally, we thank all contributors whose work has directly and indirectly impacted `CUDACPP` development, as well as all MG5AMC authors, past and present. Computational resources were partially provided by the Calcul Intensif et Stockage de Masse (CISM) technological platform. SR and ZW acknowledge support from CERN openlab as well as the Next Generation Triggers project hosted by CERN, which is funded by the Eric and Wendy Schmidt Fund for Strategic Innovation.

Funding This work benefited from no explicit funding other than that provided as salaries for the authors.

Data Availability Statement My manuscript has associated data in a data repository

Code Availability Statement My manuscript has associated code/software in a data repository

Funded by SCOAP[3].

References

1. J. Albrecht et al., A roadmap for HEP software and computing R&D for the 2020s. Comput. Softw. Big Sci. **3**(1), 7 (2019). https://doi.org/10.1007/s41781-018-0018-8. arXiv:1712.06982 [physics.comp-ph]
2. The ATLAS Collaboration, ATLAS Software and Computing HL-LHC Roadmap. https://cds.cern.ch/record/2802918. Geneva, (2022)
3. CMS Offline Software and Computing, CMS Phase-2 Computing Model: Update Document. https://cds.cern.ch/record/2815292. Geneva, (2022)
4. A. Valassi, E. Yazgan, J. McFayden, et al., Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC. Comput. Softw. Big Sci. 5(1) (2021). Ed. by A. Valassi, E. Yazgan, and J. McFayden, p. 12. https://doi.org/10.1007/s41781-021-00055-1. arXiv:2004.13687 [hep-ph]
5. J. Kanzaki, Monte carlo integration on GPU. Eur. Phys. J. C **71**, 1559 (2011). https://doi.org/10.1140/epjc/s10052-011-1559-8. arXiv:1010.2107 [physics.comp-ph]
6. K. Hagiwara et al., Fast calculation of HELAS amplitudes using graphics processing unit (GPU). Eur. Phys. J. C **66**, 477–492 (2010). https://doi.org/10.1140/epjc/s10052-010-1276-8. arXiv:0908.4403 [physics.comp-ph]
7. K. Hagiwara et al., Fast computation of MadGraph amplitudes on graphics processing unit (GPU). Eur. Phys. J. C **73**, 2608 (2013). https://doi.org/10.1140/epjc/s10052-013-2608-2. arXiv:1305.0708 [physics.comp-ph]
8. E. Bothmann et al., Many-gluon tree amplitudes on modern GPUs: A case study for novel event generators. SciPost Phys. Codeb. **2022**, 3 (2022). https://doi.org/10.21468/SciPostPhysCodeb.3. arXiv:2106.06507 [hep-ph]
9. S. Carrazza et al., MadFlow: automating Monte Carlo simulation on GPU for particle physics processes. Eur. Phys. J. C **81**(7), 656 (2021). https://doi.org/10.1140/epjc/s10052-021-09443-8. arXiv:2106.10279 [physics.comp-ph]
10. E. Bothmann et al., A portable parton-level event generator for the high-luminosity LHC. SciPost Phys. **17**(3), 081 (2024). https://doi.org/10.21468/SciPostPhys.17.3.081. arXiv:2311.06198 [hep-ph]
11. J.M. Cruz-Martinez, G. De Laurentis, M. Pellen, Accelerating Berends-Giele recursion for gluons in arbitrary dimensions over finite fields. Eur. Phys. J. C **85**(5), 590 (2025). https://doi.org/10.1140/epjc/s10052-025-14318-3. arXiv:2502.07060 [hep-ph]
12. A. Valassi et al., Design and engineering of a simplified workflow execution for the MG5aMC event generator on GPUs and vector CPUs. EPJ Web Conf. **251**, 03045 (2021). https://doi.org/10.1051/epjconf/202125103045. arXiv:2106.12631 [physics.comp-ph]
13. A. Valassi et al., Developments in Performance and Portability for MadGraph5_aMC@NLO. PoS ICHEP2022 (2022), p. 212. https://doi.org/10.22323/1.414.0212. arXiv:2210.11122 [physics.comp-ph]
14. A. Valassi et al., Speeding up Madgraph5 aMC@NLO through CPU vectorization and GPU offloading: towards a first alpha release. In: 21th International Workshop on Advanced Computing and Analysis Techniques in Physics Research: AI meets Reality. (2023). arXiv:2303.18244 [physics.comp-ph]
15. S. Hageboeck et al., Madgraph5_aMC@NLO on GPUs and vector CPUs experience with the first alpha release. EPJ Web Conf. **295**, 11013 (2024). https://doi.org/10.1051/epjconf/202429511013. arXiv:2312.02898 [physics.comp-ph]

16. A. Valassi et al. Madgraph on GPUs and vector CPUs: towards production (The 5-year journey to the first LO release CUDACPP v1.00.00). In: 27th International Conference on Computing in High Energy and Nuclear Physics. (2025). arXiv: arXiv:2503.21935 [physics.comp-ph]
17. S. Hageböck et al., Data-parallel leading-order event generation in MadGraph5 _aMC@NLO. arXiv:2507.21039 [hep-ph]. (2025)
18. J. Alwall et al., A standard format for les houches event files. Comput. Phys. Commun. **176**, 300–304 (2007). https://doi.org/10.1016/j.cpc.2006.11.010. arXiv:hep-ph/0609017
19. J.M. Butterworth et al., THE TOOLS AND MONTE CARLO WORKING GROUP Summary Report from the Les Houches 2009 Workshop on TeV Colliders. 6th Les Houches Workshop on Physics at TeV Colliders. Mar. 2010. arXiv:1003.1643 [hep-ph]
20. J. Andersen et al., Les Houches 2013: Physics at TeV Colliders: Standard Model Working Group Report. Tech. rep. Comments: Proceedings of the Standard Model Working Group of the 2013 Les Houches Workshop, Physics at TeV Colliders, Les houches 3-21 June 2013. 200 pages. 2013 Les Houches Workshop, 2014. arXiv:1405.1067. http://cds.cern.ch/record/1699963
21. J. Alwall et al., The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations. JHEP **07**, 079 (2014). https://doi.org/10.1007/JHEP07(2014)079. arXiv:1405.0301 [hep-ph]
22. O. Mattelaer, On the maximal use of monte carlo samples: re-weighting events at NLO accuracy. Eur. Phys. J. C **76**(12), 674 (2016). https://doi.org/10.1140/epjc/s10052-016-4533-7. arXiv:1607.00763 [hep-ph]
23. I. Brivio, Y. Jiang, M. Trott, The SMEFTsim package, theory and tools. JHEP **12**, 070 (2017). https://doi.org/10.1007/JHEP12(2017)070. arXiv:1709.06492 [hep-ph]
24. I. Brivio, SMEFTsim 3.0 – a practical guide. JHEP **04**, 073 (2021). https://doi.org/10.1007/JHEP04(2021)073. arXiv:2012.11343 [hep-ph]
25. M. Dobbs, J.B. Hansen, The HepMC C++ Monte Carlo event record for High Energy Physics. Comput. Phys. Commun. **134**(1), 41–46 (2001). https://doi.org/10.1016/S0010-4655(00)00189-2
26. A. Buckley et al., The HepMC3 event record library for Monte Carlo event generators. Comput. Phys. Commun. **260**, 107310 (2021). https://doi.org/10.1016/j.cpc.2020.107310. arXiv:1912.08005 [hep-ph]
27. A. Verbytskyi et al., HepMC3 event record library for monte carlo event generators. J. Phys: Conf. Ser. **1525**(1), 012017 (2020). https://doi.org/10.1088/1742-6596/1525/1/012017
28. The HDF Group, Hierarchical Data Format, version 5. HDF5 Online Documentation. https://github.com/HDFGroup/hdf5
29. S. Höche, S. Prestel, H. Schulz, Simulation of vector boson plus many jet final states at the high luminosity LHC. Phys. Rev. D **100**(1), 014024 (2019). https://doi.org/10.1103/PhysRevD.100.014024. arXiv:1905.05120 [hep-ph]
30. E. Bothmann et al., Efficient precision simulation of processes with many-jet final states at the LHC. Phys. Rev. D **109**(1), 014013 (2024). https://doi.org/10.1103/PhysRevD.109.014013. arXiv:2309.13154 [hep-ph]
31. E. Boos et al., Generic User Process Interface for Event Generators. In: 2nd Les Houches Workshop on Physics at TeV Colliders. (2001). arXiv:hep-ph/0109068
32. Intel Corporation, Intel®Xeon®Gold 5118 Processor. https://www.intel.com/content/www/us/en/products/sku/120473/intel-xeon-gold-5118-processor-16-5m-cache-2-30-ghz/specifications.htmlOnline product listing. (2017)
33. G. Isidori, F. Wilsch, D. Wyler, The standard model effective field theory at work. Rev. Mod. Phys. **96**(1), 015006 (2024). https://doi.org/10.1103/RevModPhys.96.015006. arXiv:2303.16922 [hep-ph]
34. Exxact Corporation, Comparing Blackwell vs Hopper: B200 & B100 vs H200 & H100: Exxact blog. (2024). https://www.exxactcorp.com/blog/hpc/comparing-nvidia-tensor-core-gpus
35. T.P. Morgan, Sizing Up Compute Engines For HPC Work At 64-Bit Precision. (2025). https://www.nextplatform.com/2025/02/20/sizing-up-compute-engines-for-hpc-work-at-64-bit-precision/
36. Z. Wettersten, Rex & teaRex: Libraries for efficient access and generic reweighting of parton-level events (v1.0.0). Version 1.0.0. (2025). https://doi.org/10.5281/zenodo.17573568
37. European Organization For Nuclear Research and OpenAIRE. Zenodo. en. (2013). https://doi.org/10.25495/7GXK-RD71, https://www.zenodo.org/