Merlin++, a flexible and feature-rich accelerator physics and particle tracking library

Robert B. Appleby[a,b], Roger J. Barlow[c,*], Dirk Krücker[d], James Molson[e], Scott Rowan[c], Sam Tygier[a], Haroon Rafique[e], Nicholas Walker[d], Andrzej Wolski[f,b]

[a]*The University of Manchester, Oxford Rd, Manchester M13 9PL, UK*
[b]*The Cockcroft Institute, Keckwick Ln, Daresbury, Warrington, WA4 4AD, UK*
[c]*The University of Huddersfield, Queensgate, Huddersfield, HD1 3DH, UK*
[d]*DESY, Notkestraße 85, D22607 Hamburg, Germany*
[e]*CERN, CH1211 Genève 23, Switzerland*
[f]*The University of Liverpool, Liverpool L69 7ZX, UK*

Abstract

Merlin++ is a C++ charged-particle tracking library developed for the simulation and analysis of complex beam dynamics within high energy particle accelerators. Accurate simulation and analysis of particle dynamics is an essential part of the design of new particle accelerators, and for the optimization of existing ones. Merlin++ is a feature-full library with focus on long-term tracking studies. A user may simulate distributions of protons or electrons in either single particle or sliced macro-particle bunches. The tracking code includes both straight and curvilinear coordinate systems allowing for the simulation of either linear or circular accelerator lattice designs, and uses a fast and accurate explicit symplectic integrator. Physics processes for common design studies have been implemented, including RF cavity acceleration, synchrotron radiation damping, on-line physical aperture checks and collimation, proton scattering, wakefield simulation, and spin-tracking. Merlin++ was written using C++ object orientated design practices and has been optimized for speed using multicore processors. This article presents an account of the program, including its functionality and guidance for use.

Program Summary
Program Title: Merlin++
Program Files DOI: 10.5281/zenodo.3700155
Licensing provisions: GPLv2+
Programming language: C++
Nature of problem: Complexity of particle accelerators beam dynamics over extensive tracking distances.
Solution method: Long-term particle accelerator and tracking simulations utilizing explicit symplectic integrators.
Additional comments: For further information see github.com/Merlin-Collaboration

Keywords: Particle Accelerator, Proton, Electron, Sliced Macroparticle, LHC, Synchrotron, Linac, Tracking, Symplectic, Spin Tracking, Wakefield, Collimation, Scattering

1. Introduction

Modern high energy particle accelerators are large and complex, and beam dynamics simulations are therefore fundamental to their design and optimization.

A charged particle accelerator is designed as a lattice structure of electromagnetic accelerating and/or focusing components. An initial design may consist of standard arrangements of common elements such as transverse-magnetic (TM) mode radio-frequency (RF) resonating cavities to transfer energy and accelerate particles, and dipole and quadrupole electromagnets to guide and focus the beam. A more refined design is then achieved through an iterative process of beam dynamics simulations wherein lattice element parameters are altered and higher-order magnetic components such as sextupoles and octupoles are included where necessary to account for non-linear instabilities. Specific lattice sections may be designed for specific purposes: for example, a lattice subsection of bending dipoles may be constructed symmetrically in a double achromat so that there is zero energy dispersion in the beam at either end, or a triplet of strong quadrupole magnets may be used to focus the beam at a collision point.

Successful operation of an accelerator relies on beam stability and field synchronisation. For example, in linear accelerators, with aligned RF cavities in series, synchronization of AC voltage signals with incident particles is vital in achieving acceleration. For circular accelerators, avoiding disruptive resonances is important to ensure long-term stability, as losses may only develop over thousands or even millions of turns. Due to the scale and complexity of modern accelerator lattices, accurate beam dynamics simulations are crucial in optimizing the dynamic aperture and beam lifetime.

Real world lattice elements will not match the optimal de-

[*] Corresponding author.
E-mail address: roger.barlow@cern.ch (R. J. Barlow)

sign exactly, being subject to alignment and field strength errors. Simulations are also necessary to ensure that any loss of performance due to such variation falls within allowable limits. A simulation is then performed not just once but many times, so the program needs to be optimized for speed.

Merlin++, described in this article, is such a simulation program. After a summary of existing tracking codes and how Merlin++ compares with them in Section 2 the physical processes modelled are described in Section 3 and software issues in Section 4. Section 5 contains results from benchmarking Merlin++ against data and other codes, and Section 6 outlines the measures used to optimize the speed of the program. Section 7 shows a potential user how to get started, and finally some suggestions for possible future development are given in Section 8.

This article is intended primarily for readers who need to simulate an accelerator and are considering whether Merlin++ will provide what they require: a knowledge of basic accelerator concepts is therefore assumed. Any other readers (perhaps software specialists) can, if interested, find explanations in any basic accelerator textbook, such as [1].

2. Tracking Codes and Merlin++

In recent decades, significant advances have been made in the accuracy and stability of mathematical methods used by particle tracking codes. Historically, so-called matrix codes such as TURTLE[2], which used Brown and Rothacker's Transport [3], used the thin-lens approximation with transfer matrices propagating a particle through an accelerator lattice along a path close to a reference orbit. Such codes are inherently limited due to truncation of the maps to a specific order of non-linearity, typically 3rd-order. More accurate codes, such as Cosy-Infinity [4] which uses differential algebra to generate maps to arbitrary order, can be used for tracking with a distribution of particles with varying particle energies/orbits, however, this method very quickly becomes impractical for large lattices and long simulations due to computational limitations. Ruth [5] proposed a more practical approach, involving the derivation of explicit equations for each component from integrable component-specific Hamiltonians. This allows for accurate stepwise propagation through each element. Such integrators are possible due to most common accelerator component Hamiltonians being integrable, either directly or when split. More advanced Hamiltonian splitting methods for various components were further developed by Forest and Ruth [6], Yoshida [7], Wu, Forest and Robin[8] and Laskar and Robutel [9].

Various beam dynamics and accelerator physics codes have been developed over the years to assist in the accelerator design process. For example MAD [10], is widely used for single particle dynamics simulations, Bmad [11], to study non-linearities and collective effects, FLUKA [12] for detailed particle-matter interactions, and there are many more. Historically, whenever the scope of a research study exceeded that of existing codes, designers had to create their own code base. The programming languages and practices utilized have not generally been sustainable, so many tracking codes have been developed, used and abandoned. Due to lack of an alternative some code bases are maintained even with known core design issues and/or being built in an outdated or inefficient language. Moreover a new accelerator simulation will require consideration not only of magnets and RF cavities but of other physical processes which influence particles, such as synchrotron radiation and collimation. A typical existing tracking code is limited in the additional processes that may be included in the simulation, and in many cases, due to being later designed as an addition/attachment to a legacy code base, in these codes such features have computationally expensive implementations.

Merlin++ was therefore architecturally designed as expandable and general-purpose. This was achieved using the C++ language. C++ is a high-level object-orientated language which crucially does not compromise low-level access and optimization features. Utilizing the full feature set of the language, from C++11 and beyond, such as generic programming, polymorphism and memory allocation, as well as cutting-edge technologies such as move semantics and native multi-threading, Merlin++ is designed from the ground up to be accessible, modular and fast.

Merlin++ has been used and developed for some years and many additional features and functionality have easily been incorporated by new user-developers without affecting the existing code. It has grown to be a powerful and feature-rich general purpose accelerator physics library. An example of a process added to Merlin++ in recent years is the collimation and scattering feature set, including several different scattering process such as multiple-Coulomb and Rutherford scattering as well as models for elastic proton-nucleus and single diffractive proton-nucleon interactions. Figure 1 shows an example: the Merlin++ output of particles interacting in the collimation region of the LHC. It shows how several halo particles have undergone scattering in the collimator material, but have not been absorbed and continued to propagate and subsequently impacted other components of the machine. This type of simulation is important for machine protection studies. These new physics processes were readily incorporated into the existing framework.

3. Overview of Merlin++

Merlin++ [13] is a C++ physics library for high energy particle accelerator systems design and particle tracking simulations. Its functionality is similar to that of MAD, but with many more features, and is comparable in many ways to SixTrack [14]. Over the years, Merlin++ has been extended and applied to a variety of use-cases, such as ILC beamline and damping rings studies [15, 16, 17], NLC depolarisation/spin-tracking [18], and LHC collimation [19, 20, 21, 22, 23, 24]. Currently, Merlin++ remains under active development and use for HL-LHC, FCC [25] and SppC collimation studies [26]. As a result, the code base has grown to become one of the most feature-rich and functionally capable tracking codes available.

It provides a 6-dimensional phase-space tracking library with multiple tracking integrators, including symplectic and 1$^{\text{st}}$ and

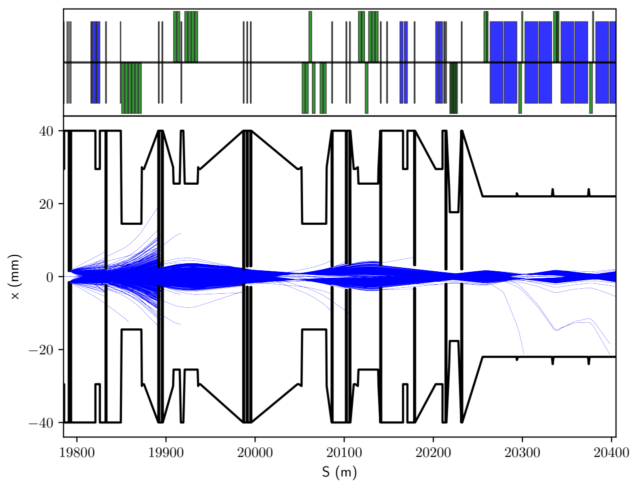

Figure 1: Scattered particles in the collimation region of the LHC. Upper plot shows the arrangement of dipoles (blue), quadrupoles (green) and collimators (black). Lower plot shows the machine aperture (black) and proton tracks (blue).

2$^{\text{nd}}$-order transport integrators. Bunches of protons or electrons can be tracked along either linear or circular accelerator lattice structures. Particle bunches may be comprised of a distribution of single particles or a collection of sliced macro-particles. Optional physics processes are also provided, including RF cavity acceleration, synchrotron radiation, on-line physical aperture checks, collimation, proton diffractive scattering, wakefield simulation and spin-tracking.

3.1. Architecture and Design Philosophy

Merlin++'s structure was developed with two fundamental design philosophies:

1. To be functional and of high performance while being easy to learn and flexible.
2. To be comprised of loosely-coupled and independently maintainable modules.

Accordingly it consists of four loosely-coupled and extensible modules:

1. Lattice Structure
2. Beam Parameters
3. Particle Tracking
4. Physics Processes

A graphical depiction of a simple particle tracking simulation based on the Merlin++ architecture is shown in Figure 2. The user program defines how the accelerator model and beam are created, and then calls the tracker to transport the particles through the lattice, applying requested physics processes at each stage.

To ensure a maintainable, modular and performant architecture, the C++ language was chosen as it contains powerful high-level OOD features such as templates and polymorphism, while also being able to manipulate low-level code to optimize program speed and memory access. Its is also a very widely taught programming language, with many physics and engineering students being familiar with it by the time they graduate. (Similar tracking codes written in FORTRAN77 and/or FORTRAN90 have a long history of usability and maintainability issues.) It was also decided that Merlin++ would remain a library rather than a stand-alone program to allow users to write use-case specific programs. This avoids the need to learn a domain-specific language. It does require users to compile and link their program with the library before running but this is straightforward and is detailed in accompanying documentation. It can be compiled and run in a terminal window and/or within an IDE such as Eclipse CDT [27]. Users can add new components and features and/or combine Merlin++ with custom or third party code, such as ROOT [28]. Merlin++ uses the CMake [29] build and test package for compilation and for running tests.

3.2. Lattice Structure

The lattice structure can be either be defined by specific calls in a user program, or imported from a MAD output file. The design follows the core concept of models, frames, elements and components. A top-down perspective would see an `AcceleratorModel` class being constructed to contain a `LatticeFrame` - itself comprised of either contiguous sections or a complete `Beamline` or `Ring` (defined model types). A frame is a geometrical construct, a local coordinate frame, but one may consider it as a hollow structure which outlines the lattice structure. Each `Beamline` or `Ring` section contains within it a number of `ComponentFrames`, providing a frame for each element. A `ComponentFrame` hosts a `ModelElement` which may or may not be a pre-defined `AcceleratorComponent`, with aperture, electromagnetic field, and geometry properties. Component classes, `SimpleDrift, SectorBend, Quadrupole` etc, are derived classes of `AcceleratorComponent`. The generic `ModelElement` root class allows users to define and seamlessly integrate additional use-case specific components such as power supplies, klystrons etc. Tracking simulations use provided iterators to cycle through each `ModelElement` to call location-specific `AcceleratorComponent` properties.

3.3. Beam Parameters

Merlin++ has a generalised concept of particle bunches, with separation between high level parameters of a beam and how it is represented in a given simulation. The `BeamData` class holds the parameters of the envelope of the beam at a given point (usually the start) of an accelerator, for example the centroid position, emittance and Courant-Snyder parameters. Typically these are set from the values obtained by the `LatticeFunctionTable` as described in section 3.4. From these parameters a representation of a bunch of particles is created. For example, a `ParticleBunch` can be defined by a collection of `Particle` phase-space coordinates, however, it is also possible to define the bunch by its moments using `SMPBunch` (Sliced Macro-particle) class. If spin coordinates are required a `SpinParticleBunch` can be used, as described in section 3.6.4.

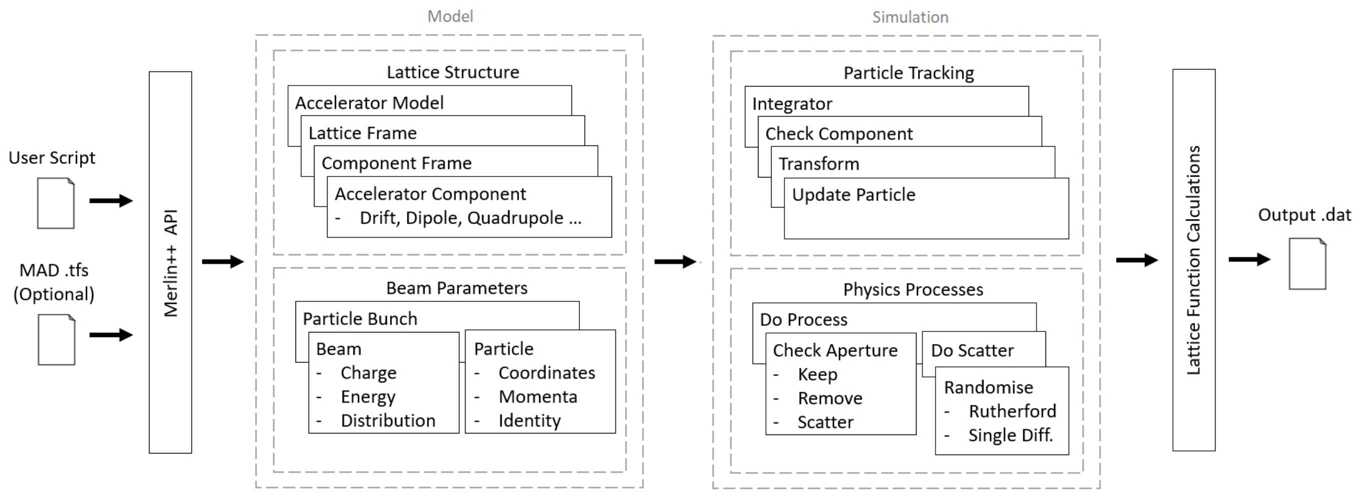

Figure 2: High-level architectural view of Merlin++ lattice construction and particle bunch tracking.

In a `ParticleBunch` each `Particle` is defined by a 6-D `PSvector` (phase-space vector) of phase-space coordinates and momenta, as defined in Appendix A. These dynamical variables are updated at each step during tracking. By convention, the first particle in a bunch is placed on the reference orbit. Some physics processes use this reference particle for alignment.

A `ParticleBunch` can be created by several methods. An empty bunch can be created and particles then added to it with repeated calls to `ParticleBunch::push_back()`; this can be useful when very specific initial coordinates are required. Or the initial coordinates can be read from a file using `ParticleBunch::Input()`, for example to import a bunch from another simulation code. Most typically a bunch with some standard distribution is required, and for this the `ParticleBunch` constructor is called with a `ParticleDistributionGenerator` and a `BeamData`. Distribution generators for several common beam distributions including normal (Gaussian), flat and various halo models are available, and additional distributions can be added by the user. The distribution options originally create coordinates in a normalized space. Then, using beam parameters stored in a `BeamData` object the coordinates are transformed by Courant-Snyder parameters to produce the final particle bunch. Phase-space distributions of normal and halo bunches are shown in Figure 3.

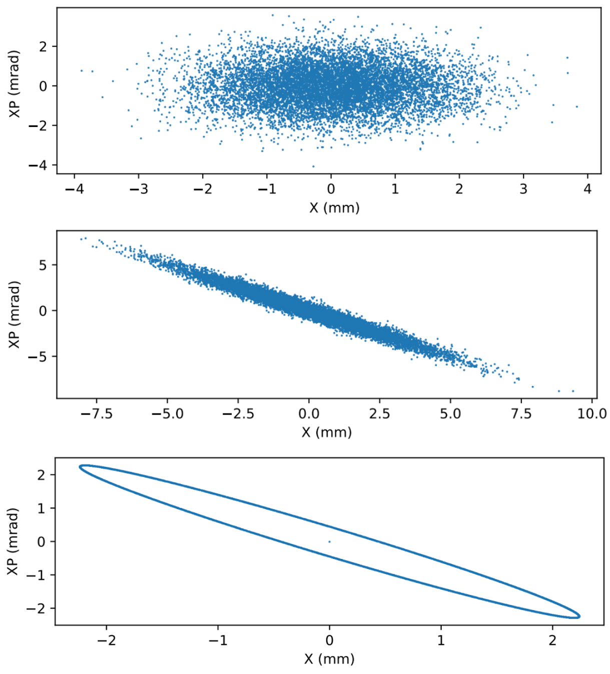

Figure 3: Bunch distribution profile in the x-x' plane. Initially the bunch is created in normalized coordinates (top), then transformed by Courant-Snyder parameters (middle). A distribution from the halo option, set at one sigma, is also shown (bottom).

3.4. Calculation of Lattice Functions

A common analysis problem is to calculate the lattice functions, also known as the Courant-Snyder or Twiss parameters, which determine the stability of a periodic lattice and are also relevant for transfer lines. Merlin++ implements a normal form approach method of calculation following Wolski [30, 31]. The closed orbit is calculated as usual and the 1-turn matrix $\mathbf{M}$ is evaluated at an arbitrary point by tracking single particles with small displacements from the closed orbit. If the orbit is stable, then $\mathbf{M}$ has eigenvalues $e^{\pm i\mu_k}$, with μ_k real, where the index k runs from 1 to 3 and can be identified, if the coupling is not too strong, with motion in the horizontal, vertical and longitudinal directions. A related matrix, $\mathbf{\Sigma}_{ij} = \left\langle x_i x_j \right\rangle$ which describes matched bunches (*i.e.* which do not change after a complete turn: $\mathbf{M\Sigma\tilde{M}} = \mathbf{\Sigma}$) can be written as the sum of three matrices $\sum \epsilon_k \mathbf{B}_k$, where ϵ_k are the emittances and elements of the 6×6 matrices $\mathbf{B}_k$ and map on to the lattice parameters in the absence of coupling. From the eigenvectors of $\mathbf{M}$, a matrix $\mathbf{N}$ can be constructed which transforms the general 6-D turn by turn motion into three normal 2-D circular forms, and the elements of $\mathbf{N}$ give the elements of $\mathbf{B}$. Subsequently, after each of the $\mathbf{B}_k$ matrices have been determined for an arbitrary starting point, they can be tracked along the lattice and the corresponding lattice functions values can be extracted at each point. If there is no coupling then the Courant-Snyder functions can be obtained from the appropriate element of the appropriate matrix, *e.g.* (in form $\mathbf{B}^k_{i,j}$) $\alpha_x = -B^1_{12}$, $\alpha_y = -B^2_{34}$, $\beta_x = B^1_{11}$, *etc.* Dispersion is calculated by $\eta_x = B^3_{16}/B^3_{66}$ and $\eta_y = B^3_{36}/B^3_{66}$. Merlin++ keeps track of these indices using instances of the `LatticeFunction` class. If requested, values are stored in a `LatticeFunctionTable`. Entries are requested by `AddFunction(i,j,k)` method, where i, j, k are the corresponding matrix elements. For simplicity, a `UseDefaultFunctions` function is provided which calculates and stores 10 common functions: $x, p_x, y, p_y, ct, \delta, \beta_x, -\alpha_x, \beta_y, -\alpha_y$. These functions can then be readily plotted by a `python` script, `gnuplot` or a similar plotting program.

3.5. *Non-linearities and Perturbation*

In engineering operationally stable accelerators, one must take into account non-linearities and the resultant instabilities. Such effects may arise as a direct result of a high-order magnetic component or from collective dynamics within a particle bunch. Utilizing Merlin++'s extensive lattice function capabilities, a user may isolate and monitor individual components or lattice regions to analyse non-linearities and the long-term affect on stability. For example, if a perturbation effect of magnetic components occurs when operating near a specific order-related phase advance 2π integer fraction, repeated effects accumulate. Figure 4 shows a simple example of adding a sextupole to the end of an otherwise stable periodic FODO lattice cell. As shown, varying the phase advance (by adjusting the quadrupole strengths) can result in significant instabilities. Such analysis is important in optimizing the dynamic aperture.

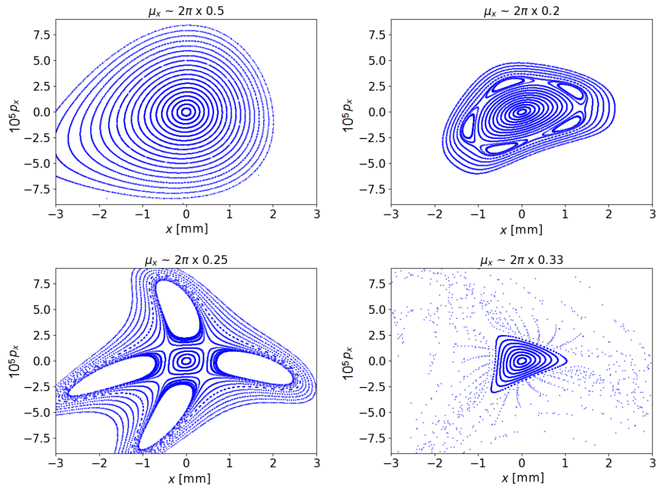

Figure 4: The effect on the horizontal parameters of adding a constant length/field sextupole to the end of a periodic FODO lattice cell. Shown are half, third, quarter and fifth integer phase advances μ_x.

3.6. *Particle Tracking Integrators*

Merlin++ provides various integrator options for particle tracking. The `SYMPLECTIC` integrator is the default and is what advised for simulations of large numbers of turns. `TRANSPORT` and `THIN_LENS` are also available for bench-marking against other software packages. In each case a given particle, defined by its `PSvector` variables is subject to an explicit transformation as it propagates through a component.

3.6.1. *The Symplectic Integrator*

Any numerical integration inevitably introduces inaccuracy. A symplectic integrator preserves the Hamiltonian, though it is not necessarily the most accurate integrator for some individual particle parameter.

From the accelerator Hamiltonian, individual component integrators are derived by first defining the component specific electromagnetic fields and simplifying the Hamiltonian, accordingly. Using Hamilton's equations, one establishes and solves the equations of motions to arrive at a transfer map and/or explicit equations for each dynamical variable. Not all component Hamiltonians are integrable and are split into drifts and kicks: $H = H_d + H_k$, where the drafts are independently integrable and the kicks requireapproximation, commonly of 2nd-order, and may use a small angle (paraxial) approximation,

The loss in accuracy can be minimised by splitting the Hamiltonian further, *e.g.* $H = H_d/2 + H_k + H_d/2$. This splitting method can be extrapolated to much higher-order [7] using Lie algebra manipulation. For the drift component, one may integrate the exact Hamiltonian. Furthermore, Forest has shown that it is possible to derive an exact solution for rectangular and sector bend dipole magnet components [32]. This allows for a considerable increase in accuracy and use of the splitting method with the exact drift and sector bend components forms the basis of the Merlin++ symplectic integrator and makes it a suitable software package for long-term tracking and beam-lifetime studies. Details are given in Appendix B.

3.6.2. *RF Cavity Structures*

Merlin++ has a number of RF options, including both travelling and standing wave structures. Simplified structures are also available which have no transverse focusing. Transverse magnetic mode (TM$_{010}$) structures simulate acceleration by by change in the particle momentum:

$$\Delta\delta = \Delta p_z - \Delta p_{z,\mathrm{ref}} = -\frac{qV}{P_0 c}[\cos(\phi + \Delta\phi) - \cos(\phi)]. \quad (1)$$

where V is the cavity voltage and ϕ is the phase. This can be followed by a redefinition of the reference momentum to avoid excessive values of δ. Where required, transverse focusing is included, for the appropriate RF wave structure, in accordance with Rosenzweig and Serafini [33], such that

$$\mathbf{M} = \begin{pmatrix} \cos(\alpha) - \sqrt{2}\cos(\phi)\sin(\alpha) & \frac{\gamma_\mathrm{i}}{\gamma_\mathrm{f}}\sqrt{8}\cos(\phi)\sin(\alpha) \\ -\frac{\gamma_\mathrm{i}}{\gamma_\mathrm{f}}\left(\frac{\cos(\phi)}{\sqrt{2}} + \frac{1}{\sqrt{8}\cos(\phi)}\right)\sin(\alpha) & \frac{\gamma_\mathrm{i}}{\gamma_\mathrm{f}}\left(\cos(\alpha) + \sqrt{2}\cos(\phi)\sin(\alpha)\right) \end{pmatrix}, \quad (2)$$

where $\mathbf{M}$ is the transfer map, γ_i and γ_f are the initial and final Lorentz factors and

$$\alpha = \sqrt{\frac{1}{8}\frac{1}{\cos(\phi)}} \log\left(\frac{\gamma_\mathrm{f}}{\gamma_\mathrm{i}}\right). \tag{3}$$

A `Klystron` class also allows configuration of arrays of cavities to model the RF control and power system.

In addition to the above, transverse electric (TE) mode structures have be implemented to simulate crab cavities for HL-LHC studies. The transverse kick follows

$$\Delta p_t = -\frac{eV_{cc}}{E_0}\sin(\phi + \Delta\phi), \tag{4}$$

where V_{cc} is the strength of the cavity and E_0 the reference energy. The kick strength may be dynamically altered during simulation to model a cavity failure via the `CCFailureProcess`.

3.6.3. *Sliced Macroparticle Tracking*

In situations where a user is simulating linear lattice models or does not require detailed bunch descriptions, a sliced macroparticle (SMP) model of a bunch may be used. For a `SMPBunch`, phase-space coordinates are replaced by 1st and 2nd order *moments*. In this case, the normal tracking integrator does not work and corresponding `SMPComponentTracker` and `SMPStdIntegrators` classes define specific equivalents. The SMP integrator does not fully model non-linear lattice elements, although it does include feed-down effects from them. SMP bunches allow for implementation of collective effect physics processes and have been extensively used in wakefield simulation studies.

3.6.4. *Spin Tracking*

Spin tracking was implemented to simulate electron beam depolarisation in the Next Linear Collider (NLC) Main Damping Rings [18]. Electron spin dynamics in a high-energy storage ring can be described by the Thomas-BMT equation [34, 35], where the precession of the spin vector, $\vec{S}$, follows

$$\frac{d\vec{S}}{dt} = \vec{\Omega} \times \vec{S} \tag{5}$$

where $\vec{\Omega}$ is constructed from the electromagnetic fields. Assuming the electric field is zero, $\vec{\Omega}$ can be defined, for linear coordinates, by

$$\vec{\Omega} = -\frac{e}{\gamma m}[(1 + G\gamma)\vec{B}_\perp + (1 + G)\vec{B}_\parallel] \tag{6}$$

and for curvilinear coordinates, by

$$\vec{\Omega} = -\frac{e}{\gamma m}[G\gamma\vec{B}_\perp + (1 + G)\vec{B}_\parallel] \tag{7}$$

where G is the anomalous magnetic moment (0.00115965 for electrons) and $\vec{B}_\perp$ and $\vec{B}_\parallel$ are perpendicular and parallel electron motions, respectively. In addition to tracking the above, Merlin++ also implements dipole magnet fringe fields for spin tracking as they can change the vertical component of the spin vector. The fringe field model used is

$$\int B_z ds = \frac{1}{2} y\hat{B}_y. \tag{8}$$

3.6.5. *Physical Aperture Boundaries*

A powerful tool of the Merlin++ tracking routine is physical aperture boundary checking, if a `CollimateParticleProcess` is activated. If aperture information is provided, following every discrete propagation along a beamline, a particle's coordinates are compared against the physical aperture boundaries and if it exceeds these the particle is considered lost or, if the `CollimateProtonProcess` child of `CollimateParticleProcess` is used, be subject to a physics process, such as `ScatteringProcess`. Merlin++ supports a number of aperture geometries, including `Rectangular`, `Circular`, `Elliptical`, `Rectellipse` and `Octagon`. If additional aperture shapes are required the user can create a subclass of `Aperture` and override the `CheckWithinApertureBoundaries` method. Element-specific step sizes can be defined, for example, every 50 cm or only at the beginning and end of an element. If more accurate checks are required, *e.g.* for a collimator, and the provided aperture data is less discrete than the step size, linear interpolation of aperture data is carried out prior to boundary checks.

3.7. *Physics Processes*

In many cases a user may wish to add additional physics processes for the tracker to call, *e.g.* collimation, synchrotron radiation *etc*. Physics processes are defined in the `BunchProcess` class. When a process is activated for a given simulation, the tracker will call the process at each model element. At this point, the process will determine if the physics routine should be run, for example the synchrotron radiation process may be run for a dipole, but not a drift component. A priority list is defined when multiple processes are activated such that dominant processes are called first. A number of commonly used physics processes are detailed in the follow subsections. Note, however, that processes are defined generically and are flexible in nature. As such, they may used for practical purposes beyond physics functions. For example, the `MonitorProcess` class allows the user to read and output a particle's coordinates as it traverses a set of elements.

3.7.1. *Synchrotron Radiation*

The loss in particle energy due to being accelerated in a curved path or radially with respective to the current trajectory is known as synchrotron radiation. Following Burkhardt [36], the `SynchRadParticleProcess` implements a Monte Carlo photon spectrum generator. The total number of photons and/or the total energy loss is derived from the following relation in accordance with Schwinger [37],

$$\int_0^\infty x^n K_\nu(x)dx = 2^{2n-1}\Gamma\left(\frac{1+n-\nu}{2}\right)\Gamma\left(\frac{1+n+\nu}{2}\right), \tag{9}$$

where n is the number of radiated photons, ν the particle velocity and K_ν is a modified Bessel function. The Γ function has the useful property, $\Gamma(x)\Gamma(1-x) = \pi/\sin(\pi x)$. Merlin++ allows for the loss of energy to be taken into account by two different methods: first, by calculation and modification of individual particle momenta, leaving the reference momentum unchanged; second, by determination of the mean energy loss, which is then applied to the reference momentum and individual particles, accordingly. Merlin++ does not currently track the produced photons, although an implementation would be possible.

3.7.2. *Collimation and Scattering*

Controlled removal of outer particles in unstable orbits is common practice in accelerator operations. This process of active collimation involves installation of high energy-capacity collimators with adjustable mechanical jaws designed to actively reduce the local physical aperture and absorb particles that would otherwise be lost in more vulnerable components. Collimators are typically installed at high β locations along a lattice structure.

At high beam energies the collimator will not absorb the complete energy of a particle, and of any secondary particles it may produce. Systems of primary (spoiler) and secondary (absorber) collimators are used. The simulation of such complete systems requires detailed modelling, and is done by programs such as FLUKA [38, 12] and Geant4 [39] (including BDSIM [40]) at a level of detail which is not compatible with a program for simulating accelerator optics, and Merlin++ does not attempt to provide a fine-grained description of such showering processes. However it is appropriate for an accelerator optics simulation to consider particles which undergo a single scatter in a collimator, experiencing a small deflection and loss of energy, and continue to pass though the lattice, eventually being lost at some remote location – indeed, perhaps after experiencing several turns through the lattice.

The `Collimator` class provides functionality to do this, being able to model collimators with various geometries (including a one-sided jaw), and/or alignment errors, and also specific material properties, including composite materials, as described in section 7.5.2.

The `CollimateParticleProcess` implements a series of checks to determine if a particle should be lost and/or scattered. While propagating through a `Collimator` model element, if a particle is found to exceed adjusted physical aperture boundaries, but remains within the original beam-pipe boundaries, the particle is considered scattered. To model such an event, the `ScatteringProcess` class implements a number of possible scattering options, selected randomly according to their probability. Absorption (which is taken as including inelastic scattering and the start of a particle shower) is one of these; others are elastic and quasi-elastic (diffractive) scattering. Scattering options include common models for multiple coulomb, inelastic proton-nucleus, and Rutherford scattering. Moreover, Merlin++ implements uniquely developed models for elastic proton-nucleus and single diffractive proton-nucleon scattering which covers the full kinematical range of LHC energies. Full details are given in [41].

The model for elastic and single diffractive scattering incorporates both Coulomb and nuclear amplitudes as well as the interference between the two. For elastic scattering, four Regge trajectories, the hard Pomeron, soft Pomeron, the f_2 and a_2 trajectory and the ω and ρ trajectory are used for both Coulomb and nuclear amplitudes. For nuclear amplitudes an additional triple-gluon exchange is included. The full model of the differential cross-section for elastic scattering follows

$$\begin{aligned}\frac{d\sigma}{dt} = \pi[A_c(s,t)]^2 + \frac{1}{4\pi}(\Re[A_n(s,t)]^2 + \\ \Im[A_n(s,t)]^2) + (\rho + \alpha_{\mathrm{em}}\phi)A_c(s,t)\Im[A_n(s,t)], \end{aligned}\quad (10)$$

where σ is the interaction cross-section, s and t are invariants of motion, α_{em} the Regge trajectory, ϕ the Coloumb phase, ρ the ratio of real and imaginary components of the nuclear term and A_c and A_n are Coulomb and nuclear amplitudes, respectively.

The single diffractive model follows the triple-Regge description of high-mass diffractive dissociation. The implemented double differential cross section for high missing mass is

$$\begin{aligned}\frac{\partial^2\sigma^{\mathrm{HM}}}{\partial t\partial\xi}(\xi,t,s) =& g_{\mathbb{PPP}}(t)s^{\alpha_{\mathbb{P}}(0)-1}\xi^{\alpha_{\mathbb{P}}(0)-2\alpha_{\mathbb{P}}(t)}+ \\ & g_{\mathbb{PPR}}(t)s^{\alpha_{\mathbb{R}}(0)-1}\xi^{\alpha_{\mathbb{R}}(0)-2\alpha_{\mathbb{P}}(t)}+ \\ & g_{\mathbb{RRP}}(t)s^{\alpha_{\mathbb{P}}(0)-1}\xi^{\alpha_{\mathbb{P}}(0)-2\alpha_{\mathbb{R}}(t)}+ \\ & g_{\mathbb{RRR}}(t)s^{\alpha_{\mathbb{R}}(0)-1}\xi^{\alpha_{\mathbb{R}}(0)-2\alpha_{\mathbb{R}}(t)}+ \\ & \frac{g^2_{\pi\pi p}}{16\pi^2}\frac{|t|}{(t-m_\pi)^2}F^2(t)\xi^{1-2\alpha_\pi}(t)\sigma_{\pi^0 p}(s\xi),\end{aligned}\quad (11)$$

where $\xi = M^2_{\mathrm{X}}/s$ with M_{X} the missing mass, $\mathbb{P}$ and $\mathbb{R}$ denote Pomeron and effective Regge trajectories, respectively, and the 5th component is the pion exchange term. For low missing mass interactions, appropriate resonances are included and the differential cross section is given by

$$\frac{\partial^2\sigma^{\mathrm{LM}}}{\partial t\partial\xi}(\xi,t,s) = \frac{\partial^2\sigma^{\mathrm{HM}}}{\partial t\partial\xi}(\xi,t,s) + \frac{\partial^2\sigma^{\mathrm{res}}}{\partial t\partial\xi}(\xi,t,s) + R_{\mathrm{m}}(\xi,t,s), \quad (12)$$

σ_{res} is the resonance region cross section and R_{m} is a resonance matching term to account for a small step when $\xi = \xi_{\mathrm{c}}$.

3.7.3. *Hollow Electron Lens*

In some cases, relatively tight collimation is desired, but must be done without significantly increasing beam impedance. A proposed solution for proton beam accelerators is the use of so-called hollow electron lenses (HEL) – a hollow cylindrical distribution of electrons aligned in the transverse plane to create a ring for a beam to pass through. R_{min} and R_{max} give the inner and outer radii of the cylinder. For a perfect cylindrical symmetric lens, inside the hollow region ($r < R_{min}$) there is no net field produced, so the core of the proton beam is unaffected. A proton at above R_{min} will feel the electric and magnetic components of the electrons at lower radii.

For opposite charges (such as an electron lens with a proton beam), the HEL deflects high amplitude particles radially inwards towards the lens centre. The cumulative effect over many turns is to increase the amplitude of these particles, driving them onto the primary collimators. This allows for more relaxed physical aperture dimensions and an overall reduced impedance. Functionality to model HELs in Merlin++ has been implemented following Rafique [42]. Assuming a radially symmetric electron distribution, the kick applied to a proton at radial position r within the HEL radial boundaries is given by

$$\theta(r) = f(r)\frac{1}{4\pi\epsilon_0 c^2}\frac{2LI(1\pm\beta_e\beta_p)}{(B\rho)_p\beta_e\beta_p}\frac{1}{r}, \tag{13}$$

where L is the HEL length, I the electron beam current, $(B\rho)_p$ the proton beam rigidity and β_e and β_p are Lorentz β factors for the electron and proton beams, respectively. The $\pm\beta_e\beta_p$ is positive if the beams propagate in opposite directions. The $f(r)$ gives the fraction of electrons enclosed at a given radius, for a uniform electron distribution it is given by:

$$f(r) = \begin{cases} 0 & \text{if } r < R_{min} \\ \frac{r^2 - R_{min}^2}{R_{max}^2 - R_{min}^2} & \text{if } R_{min} < r < R_{max} \\ 1 & \text{if } R_{max} < r \end{cases} \tag{14}$$

3.7.4. Wakefields

Merlin++ contains the ability to calculate effects of intra-bunch wakefields, both geometric wakefields produced by changes in beam pipe aperture [43] and resistive wakefields due to induced currents [44].

A particle at radial position (r',θ') induces currents in the beam pipe whose electric and magnetic fields affect a later particle at position (r,θ). These wakefield effects alter the shape of the bunch, and in some cases this can result in a serious increase in emittance. For a perfectly conducting beam pipe of constant aperture there is no wakefield effect: wakefields due to varying geometry or finite resistivity are treated separately.

In both cases wakefields are calculated using an expansion in angular modes. The `SMPBunch` (Sliced MacroParticle) version of `ParticleBunch` sorts the particles into longitudinal order - this need only be done once. For a given mode m, and defined moments $C_j^m = \sum r'^m \cos(m\theta')$ and $S_j^m = \sum r'^m \sin(m\theta')$, the combined kick (momentum imparted) to a particle in slice i trailing N_j particles in slice $j \geq i$ by a distance s_{ji} is

$$W_x = \sum_m m r^{m-1}\left(\cos(\mu\theta)\sum_j W_m(s_{ji})C_j^m + \sin(\mu\theta)\sum_j W_m(s_{ji})S_j^m\right) \tag{15}$$

$$W_y = \sum_m m r^{m-1}\left(-\sin(\mu\theta)\sum_j W_m(s_{ji})C_j^m + \cos(\mu\theta)\sum_j W_m(s_{ji})S_j^m\right) \tag{16}$$

$$W_\parallel = \sum_m W_m'(s_{ji})r^m \sum_j \left(S_j^m\cos(m\theta) + C_j^m\sin(m\theta)\right) \tag{17}$$

which uses the wake functions W_m for a given aperture geometry. These (not to be confused with the wake fields themselves) describe the effect of one particle on a later particle in a particular geometry. For example, for a perfectly conducting tapered circular collimator, with radius dimension tapering from a to b, the wake function is

$$W_m(s) = 2\left(\frac{1}{a^{2m}} - \frac{1}{b^{2m}}\right)e^{-ms/a}\Theta(s), \tag{18}$$

where s is the longitudinal interval between leading and trailing particles and $\Theta(s)$ is a unit step function (ensuring later slices do not affect earlier ones).

In Merlin++ the wakefield computation and its effect can be found to any order m - unlike most wakefield simulation programs which only consider the dipole term. The downside of this is that such wake function calculations and their implementation have only been done for circular apertures. Calculations for other shapes - of which parallel-jawed collimators are typical [45] - can only be indicative.

3.8. Use Cases Examples

Merlin++, has a relatively long development history and has been used to model and study phenomena in a wide array of accelerators. Additional functionality has often been added to support these studies. Reviewing these studies gives an overview of the features and scope of the Merlin++ code base, and the extensibility of the core code design.

3.8.1. ILC Studies

Merlin++ was originally developed at DESY to support the design and optimization of the TESLA [46] and ILC electron linacs. In addition, a stand-alone package called ILCDFS was created using the Merlin++ library, to simulate luminosity stability and model methods of dispersion free steering (DFS) [15]. ILCDFS implements a beam-based alignment technique to minimise emittance growth within in a linac. This application could be constructed using Merlin++ as it is possible to first define a reference lattice and then modified versions to take into account misalignments and magnet errors. ILCDFS also models ILC collimator wakefields, alignment errors and dynamic ground motion [16]. Moreover, the generic `ROChannel` and `RWChannel` classes were used to develop an iterative correction algorithm `DFSCorrection`. ILCDFS is provided as a stand-alone application example with the Merlin++ source code.

3.8.2. NLC Studies

Merlin++ was used in the development of the main damping rings of the NLC, which needed to preserve the 80% beam polarization provided by the source. As the program included spin tracking, as described in section 3.6.4, it could be used for the tracking studies necessary to complement semi-analytic calculations of the effects of spin resonances, including magnet misalignments and fringe field effects.

These showed that the nominal operating energy was safe from strong resonant effects, but suggested the design should allow for small energy adjustments in case a weaker resonance caused unexpected issues [18].

3.8.3. LHC Studies

Merlin++ has been used to support LHC and HL-LHC collimation projects [47, 48]. The LHC is a 27 km, 7 TeV proton synchrotron. Due to the large stored energy in the beam, it is possible that scattered or lost particles can deposit sufficient energy in the super-conducting dipoles to quench the magnets, so beam losses around the ring must be controlled. Movable collimators at locations around the LHC ring can be adjusted to clean the bunch halo in a safe manner. At the high energies used, not all protons are absorbed by the collimators and many scatter and are eventually lost elsewhere. Merlin++ can simulate LHC collimation, being able to install and move/rotate/correct model elements at any location, as well as construct and append new physics processes such as `CollimateParticleProcess` and `ScatterProcess`, detailed in Section 3.7.2. For the purposes of analysis, the final locations of lost particles are stored and loss maps can be generated. An example of a LHC collimation and loss map study is provided as an example/tutorial with the source code.

Merlin++ has been used to study many operational and planned configurations of the LHC. Examples include investigation of the effect of scattering models on losses in the cold regions [47, 41], validation of losses during the dynamic changes of optics when the beam the beams are 'squeezed' for collision, and future configurations for HL-LHC [48], and novel collimation schemes such as the Hollow Electron Lens [42].

4. Software-Specific Features

4.1. Coding Practices

Constructed in C++, powerful OOD practices are fundamental to the design and functionality of Merlin++. The original design philosophy was for Merlin++ to consist of a small number of loosely coupled, self-contained modules. This has been achieved by extensive use of inheritance and polymorphism. The IS-A/HAS-A principle and S.O.L.I.D [49] design practices have been applied to all class structures. Moreover, all classes, methods and member function prototypes are accompanied by comments formatted to be read by the class library documentation generator Doxygen [50], so users can generate Merlin++ class library documentation by use of the `make doxygen` command. The class library is also available at the software website /merlin/doxygen. As an example the `Aperture` class inheritance structure produced by doxygen is shown in Figure 5.

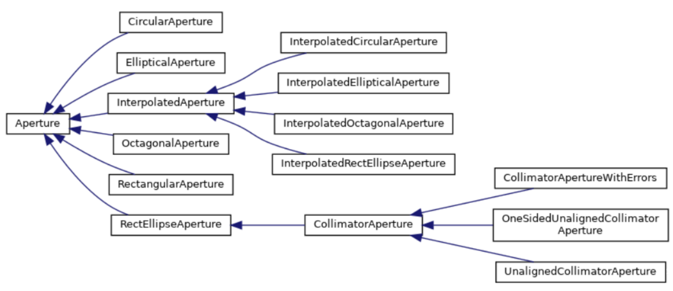
CircularAperture
EllipticalAperture
InterpolatedAperture
Aperture
OctagonalAperture
RectangularAperture
RectEllipseAperture
InterpolatedCircularAperture
InterpolatedEllipticalAperture
InterpolatedOctagonalAperture
InterpolatedRectEllipseAperture
CollimatorAperture
CollimatorApertureWithErrors
OneSidedUnalignedCollimator Aperture
UnalignedCollimatorAperture

Figure 5: Example of a class inheritance structure within Merlin++, in this case the `Aperture` class. The graphic is automatically produced by doxygen to be included in the documentation.

Generic programming via polymorphic templates is also heavily used throughout the code base and has proven vital in the usability and maintainability of Merlin++ throughout its development. For example, the template class `Matrix` is defined in `TMatrixLib.h` where access methods are defined just once. A subsequent level of abstraction is then achieved by having specific matrix types defined by `typedef Matrix<double> RealMatrix` such that developers can alter/add features more intuitively while the underlying core code remains untouched[51].

4.2. I/O Channels

A user will want specific component properties to be stored and/or dynamically altered during simulation. For this purpose, in addition to the standard library I/O, Merlin++ implements generic read-only (RO) and read-write (RW) *channels*. These are useful for iterative tuning of orbit correction algorithms. For example, the `ROChannel` may be used to read and store the output of a series of beam position monitors (BPMs) along a beamline. The collected data could then be used to calculate chromatic effects and the `RWChannel` could then be scripted to alter sextupole corrector magnets, accordingly. This can be carried out iteratively within a loop until chromaticity errors are within desired limits. The user may then also re-use the `ROChannel` to read and store sextupole component properties for future reference.

4.3. Utility functions

Like any large program Merlin++ has numerous places that need similar small pieces of functionality, such as random numbers, linear algebra, file access and string matching. As a design goal, hard dependencies on external libraries have been avoided, so where functions are missing from the C++ standard library they have been implemented within the Merlin++ codebase.

An example is the `DataTable` class inspired by structured array types in high-level languages such as the Python's numpy library [52]. A `DataTable` is designed to hold tables of data with named columns that may have different types and provides accesses and modification as well as file input and output. This is especially useful for reading the TFS format used in MAD-X, for example for lattice and aperture descriptions via the `MADInterface` class. Use of `DataTable` has reduced duplication of file parsing code in several areas of Merlin++, while improving reliability.

In some cases additions to the C++ standard library have allowed removal of utility functions from Merlin++. High quality random numbers are needed in various parts of Merlin++ including bunch creation, element misalignment, scattering, synchrotron radiation and some modes of the hollow electron lens. C++11 added a random number library, which implements the Mersenne Twister [53] algorithm and a number of distributions. Mersenne Twister is widely used in Monte Carlo simulations as it has a long period, high uniformity, low correlation between values up to high dimensions and passes many statistical tests for randomness. Switching to this standard implementation allowed removal of several large source files from Merlin++ and solved some their deficiencies (slow warm-up time, limited seed space and less well studied generator function).

4.4. *Code Testing*

Merlin++ features a suite of tests that can be run to confirm and verify correct building and functioning of the software. The tests can be run during development to prevent regressions in behaviour and quickly catch new bugs. The test suite is run automatically on each proposed change uploaded to the GitHub development platform, to confirm that the change can be successfully merged, built and run on Linux, Mac and Windows systems. In addition the tests are run on a variety of operating systems and architectures by an automated system after every commit to master, and every night.

Nightly tests are run using the CDash framework which is distributed with CMake. This provides a web interface showing recent test results. Dynamic analysis using Valgrind is also performed to flag issues such as uninitialised memory and memory leaks.

Tests include a mix of small unit tests for specific functions and larger tests of complete simulations. We are moving towards a test driven development model for new features to ensure additions function as expected.

4.5. *Developer Policy and Sustainability*

Software sustainability is described by Venters *et al.* as '*a software package's capacity to endure in changing environments*' [54]. Today, this is achievable through strict adherence to modern software engineering practices, such as those outlined by the UKs Software Sustainability Institute (SSI) on the topics of usability and maintainability [55].

Merlin++ has had a number of core developers over the years and has included people from a range of backgrounds, each with varying levels of OOD and software engineering knowledge. As such, there was a notable disparity in code quality. To ensure that it is practical for Merlin++ to be utilized for years to come, current developers have invested significant time into profiling code quality metrics and reforming/refactoring code where necessary to re-establishing a consistent level of quality [56].

Quantitative analysis, carried out with Metriculator [57], Valgrind [58] and ArchDia(R) DV8 [59], found a number of code complexity and dependency issues due to misuse or absence of appropriate OOD. Such issues limit readability and evolveability. Each issue was individually dealt with, either by simple refactoring and reformation to adherence to S.O.L.I.D. class design or a complete redesign, including implementation of appropriate design-patterns, *e.g.* leverage of decorated factory patterns for run-time type assignment to reduce the complexity of identified 'God methods' containing an abundance of if/switch statements or similar. To prevent future occurrences of quality inconsistencies a number of developer policies were changed and strictly defined. Global formatting – Uncrustify [60] – and licensing – GPL2+ [61] – changes were applied, throughout. Merlin++ is now also on the stable repository platform, GitHub [62], at github.com/Merlin-Collaboration. In addition, all code being committed to the baseline must now be reviewed and approved by at least one other developer. A developer's guide detailing the above and more is now provided with the source code.

4.6. *C++14 and Beyond...*

As C++ has been developed with successive versions over the years, our design policy has been to learn and apply new functional improvements wherever they appropriately fulfill a design purpose or simplify code. The Merlin++ 5.02 release introduced use of C++11 features, including smart pointers, lambda functions, move semantics and variadic templates. The current development branch has begun to take advantage of features from C++14 and there is investigation into the concurrency potential of the C++17 standard library parallelism for future releases. However, due to the timescale over which the code has been developed, it would be impractical to update/redesign a lot of the core legacy code structures for relatively minor performance improvements, unless profiling metrics point to performance restrictions which could be circumvented by modern implementations.

5. Model Accuracy

Merlin++ has been compared and benchmarked against a variety of related simulation packages[63, 64] and to data taken from real accelerators. These studies give us confidence in its outputs. Below we give some examples of these comparisons.

5.1. *Tracking and Optics*

MADX is widely used for design and optimization in the field of accelerator physics. It computes the optical functions of a lattice using the matrix formulation of each element, therefore providing a strong independent verification of Merlin++'s method using particle tracks. Figure 6 shows the horizontal and vertical β functions around the ATLAS interaction region in the LHC, comparing Merlin++ with MADX. The discrepancy is kept to around 1 part per million or lower even in this extreme optical configuration.

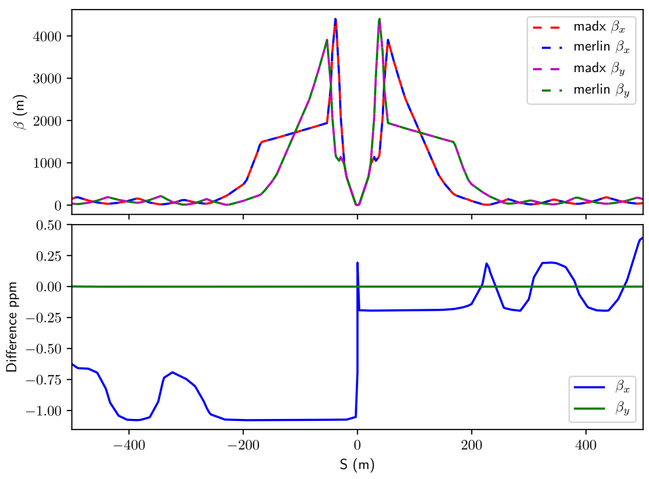

Figure 6: Comparison of β functions around the ATLAS interaction region in the LHC. The lower plot shows difference between MADX and Merlin++'s output in parts per million.

5.2. *Materials and Scattering*

Geant4 version 10.5 [39] was used to compare the implementation of Merlin++ scattering. A simple fixed target simulation was carried out in both Geant4 and Merlin++ with various materials of various depths. Two widely-used high-energy physics lists were used to gain understanding of error boundaries in the theory: QGSP_BERT and FTFP_BERT. QGSP and FTFP refer to quark-gluon-string and Fritiof string excitation models for high energy interactions, above 10 GeV, respectively. BERT refers to the Bertini cascade model for low energy particles, below 10 GeV. Figure 7 shows the comparative result for a copper-diamond (CuCD) collimator of 30 cm depth. It is clear that although simpler in nature, the Merlin++ scattering model yields similar results when compared to reputable Geant4 physics list options. The central peak – which includes most events – is identical to the eye: at large angles the difference between the two Geant models is bigger than the difference of either with the Merlin++ curve.

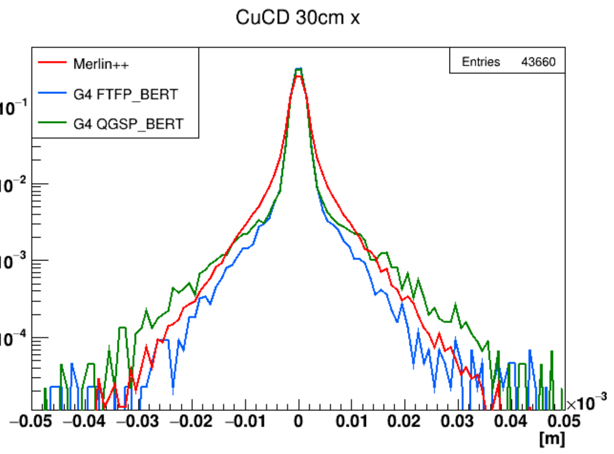

Figure 7: Comparison of fixed target horizontal scattering results between Merlin++ and Geant4. Geant4 physics lists used: QGSP_BERT and FTFP_BERT. Material used: copper-diamond (CuCD). Material depth: 30 cm.

5.3. *Collimation and Loss Maps*

The LHC is highly instrumented to monitor beam losses around the ring, with about 4000 ionization chambers making up the Beam Loss Monitor (BLM) system. In addition to their role of triggering beam dumps if loss thresholds are exceeded they log losses during all machine operation.

Merlin++ is used in the LHC collimation group for simulating these proton losses, taking into account detailed scattering models in the collimators and the aperture limits around the entire ring. These have been extensively compared to other simulations as well as to data taken during LHC operation [48].

In [48] losses were compared to BLM data under a wide range of LHC operational parameters at both 4 TeV and 6.5 TeV. Figure 8 shows a recorded and a Merlin++ simulated loss map. Merlin++ reproduces all the main loss locations as well as the collimation hierarchy, the relative losses at the primary, secondary and tertiary collimators. The article also includes comparisons of loss ratios for a range of intermediate optics points during the beam squeeze phase.

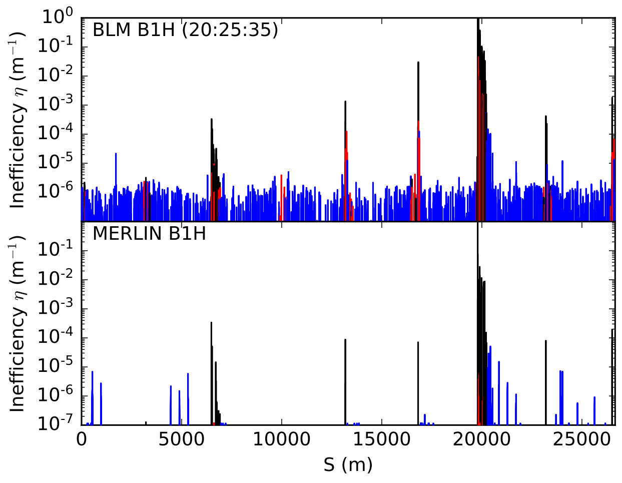

Figure 8: Beam 1 horizontal loss map from BLMs (top) and Merlin++ (bottom) at 6.5 TeV with β* of 50 cm. [48]

6. Profiling and Performance

6.1. *Fast Computation Design Features*

Merlin++ is designed to be a high performance code, capable of tracking large numbers of particles through lattices with many magnets. Data structures are designed to reduce unneeded storage overhead, and simulation code avoids unnecessary computation or data copying. Improvements are ongoing and changes are made where profiling shows that they will be effective. For example, the `RTMap` class for second order transfer maps stores only a sparse matrix of the non-zero elements to reduce unnecessary computations.

6.2. *Multi-threading*

For the most common use cases of Merlin++ only single particle dynamics are considered. This allows many Merlin++ simulations to be split into completely independent tasks, each with a subset of particles. These can be run as separate processes on one or more computer nodes. Merlin++ supports the use of Message Passing Interface (MPI) parallelism for this purpose. The output from each process can then be combined with a post processing script if needed, for example to sum the losses in each element.

Figure 9 shows the throughput from running an LHC 200 turn loss map simulation with multiple processes. The benchmark was run on a dual Intel Xeon E5-2650 v2 system with 16 cores and 32 threads using hyper threading, with 10k particles per process. The performance increase is close to the expected ideal scaling up to the 16 real cores, beyond that the gain is smaller due to shared resources in hyperthreading.

This method works well with high throughput batch systems such as HTCondor [65] which allows large clusters to be formed even from inhomogeneous and intermittently available compute nodes.

For cases where multi-particle or bunch-wide dynamics are required communication between processes must be taken into account, *e.g.* tracking sliced macro-particle bunches in wake

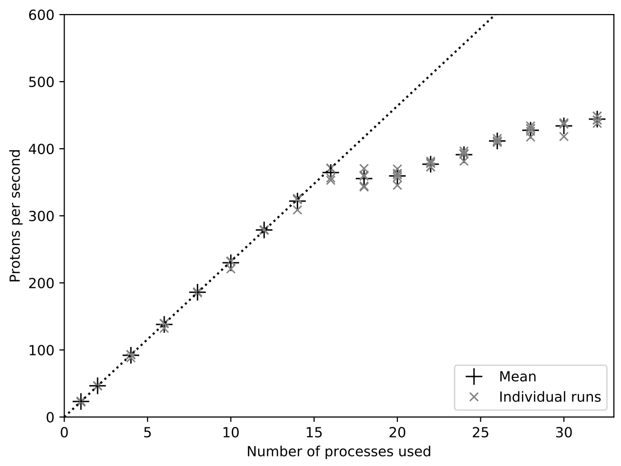

Figure 9: Performance of a benchmark test case with number of processes on a 16-core 32-thread Xeon processor. Dashed line shows ideal scaling.

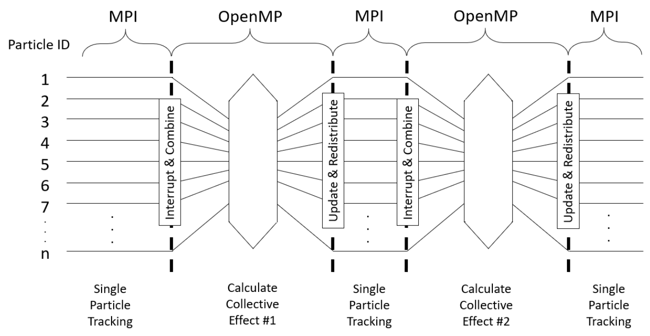

Figure 10: Graphical depiction of how Merlin++ allows the combination of both distributed single particle tracking (MPI) and calculation concurrency (OpenMP) to maximise concurrency potential in taking into account collective effects.

fields simulations. For this purpose, Merlin++ has built-in support for OpenMP parallelism. Merlin++ allows the user to define interrupts/halts for individual particle trackers at a given process stage. The user may then combine all particle data to carry out a collective bunch-wide calculation utilizing multi-threading where possible without resulting in data races. Merlin++ can then redistribute updated individual particle data to continue the tracking process. This method of parallelism for collective effects is depicted in figure 10. Performance of Merlin++ running in OpenMP mode depends strongly on the amount and frequency of inter-process communication needed for the simulation and the cluster hardware. For OpenMP use-cases, the developers are investigating the use of C++17 standard library native parallel algorithms for a future release.

7. Using Merlin++

7.1. Installation and Compilation

Merlin++ is distributed in source code form and is compiled by the user.

Current system requirements are a C++ compiler (ideally `g++`), CMake and `git`. Python is needed to run the initial tests, which though not strictly essential is strongly advised. If Eclipse is being used as an IDE then the Java Runtime Environment is needed. These are all standard and exist on most systems used for scientific calculation. The CERN ROOT package is useful but not essential. The user creates and initialises a `git` directory and clones Merlin++ into it from the repository. They then create a build directory and run `CMake` and `make` to compile the library. A test suite is run to verify a successful download, and the user can then start writing and developing their own programs.

Full (and up to date) installation instructions can be found on the Merlin++ website [66].

7.2. User Programs

A Merlin++ user program is written in C++ and linked with the Merlin++ library prior to execution. While the possibilities for a user program are large, a typical program will usually contain the following steps:

- Load an accelerator lattice description from a file or build one programmatically.
- Define or load an initial bunch of particles
- Set up a tracker with appropriate physics processes
- Run the tracker
- Perform analysis on and/or output the results.

A code snippet showing the structure of a basic Merlin++ simulation is provided in Figure 11. More advanced use-cases may include dynamic modification of model elements, interfacing with external programs and/or iteration of beam/optics parameters over multiple simulation runs. A number of examples for common cases are provided with the Merlin++ source code. Furthermore, an accompanying *'Quick-Start Guide'* provides a detailed walk-through for installation and compilation and a series of tutorials on user program development, including tracking, lattice construction/import and manipulation and physics process activation. Corresponding Python [67] output and analysis scripts are also provided for each tutorial.

7.3. Defining a Lattice

A lattice structure can either be defined manually by using the native API or automatically via importing a MAD X `.tfs` file, which is done using the `MADInterface` class. For manual construction, the user must first define a new model, which is most easily done by instantiating the `AcceleratorModelConstructor` container class and calling its member function `NewModel()`. An accelerator model can then have `AcceleratorComponents` appended via the `AppendComponent()` method which creates and assigns a new instance of a specified component. A component constructor typically requires a reference ID as well as length and field parameters, k_1, k_2, etc., *e.g.*

```
int main(int argc, char** argv)                             (a)
{
    AcceleratorModelConstructor ctor;                       (b)

    double q1_len = 1 * meter;                              (c)
    double q1_k1 = 0.5;
    ...
    ctor.AppendComponent(new Quadrupole("q1", q1_len, q1_k1));
    ctor.AppendComponent(new Drift("d1", d1_len));
    ...
    AcceleratorModel* lattice = ctor.GetModel();

    BeamData mybeam;                                        (d)
    mybeam.p0 = 1 * GeV;
    ...
    ProtonBunch* myBunch =
        new ProtonBunch(npart, NormalParticleDistributionGenerator(), mybeam);

    ofstream bunch_i("init_bunch.dat");                     (e)
    myBunch->Output(bunch_i, true);

    AcceleratorModel::RingIterator ring = lattice->GetRing();   (f)
    ParticleTracker tracker(ring, myBunch);

    tracker.Track(myBunch);

    ofstream bunch_f("final_bunch.dat");                    (g)
    myBunch->Output(bunch_f, true);
}
```

Figure 11: A simplified example of a user program. The program is standard C++ (a). First an accelerator lattice is constructed (b) which can be parametrized (c). Then the beam parameters can be defined and a particle bunch constructed (d). The initial bunch is written to disk (e), then tracked though the lattice (f) and the resulting particle coordinates written to disk (g).

`Quadrupole('q1',q1_len,q1_k1)`. To make the construction of a periodic lattice which contains multiple identical lattice cells in succession one can simply implement a series of `AppendComponent` commands in a loop. To finalise the lattice definition an instance of the `AcceleratorModel` class must be created which accepts an `AcceleratorModelConstructor` as an input, using the `GetModel()` member function. The `AcceleratorModel` class comes defined with functional iterators `::RingIterator`, to cycle through the lattice components if required for analysis. This iterator is also used by the tracker itself.

7.4. *Modifying Lattice Elements*

Specific lattice elements can be modified without having to redefine the whole lattice. Depending on how the lattice is defined the program may already have pointers to the element of interest or it can be found with searches on name or type, for example with `AcceleratorModel::ExtractTypedElements` class method. To offset a magnet's alignment, a user would extract a `MagnetMoverList`, then use the `magnet[n]->SetX()` and `magnet[n]->SetY()` member functions to alter the transverse position of magnet n. To alter the field one can use `GetFieldStrength()` method which is defined for standard magnets or directly access the field using `AcceleratorComponent::GetEMField()`. An example of lattice modification is shown in figure 12. Lattice modification can be done iteratively within a loop to achieve fine-tuning in lattice design stability.

7.5. *Defining Element Apertures*

Real-time aperture boundary checks can be used to simulate the presence and limitations of a physical vacuum pipe. Physical boundaries are assigned to specific elements using the `ApertureConfiguration` class. The class constructor accepts a MAD-style `.tfs` file input which specifies

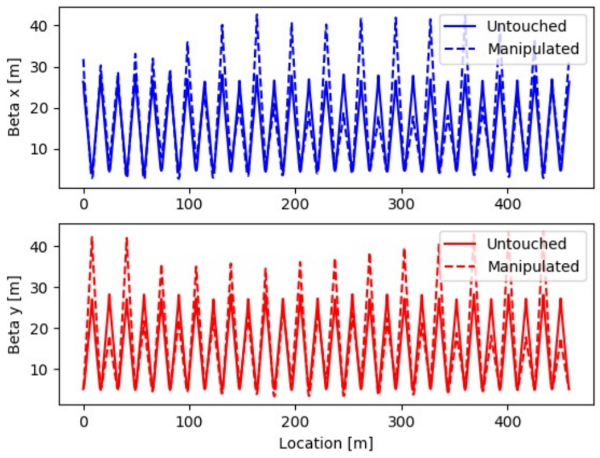

Figure 12: An example of how modifying specific lattice element position and field strengths affect the beta function. Iterative modification can be implemented to achieve automated design of stable lattices.

apertures using the MAD parameters APER_1-4. This allows the definition of multiple aperture shapes, the options being `RectangularAperture`, `CircularAperture`, `EllipticalAperture`, `RectEllipseAperture`, and `OctagonalAperture`. (If a user requires a shape other than these it would be simple for them to write an appropriate class definition.) Following successful import of the aperture parameters, the user then calls the `::ConfigureElementApertures()` method.

7.5.1. *Collimator Elements*

Collimators can be included as part of a lattice structure. To add a series of collimators at specified locations, The user provides a collimator database in a text file. This file contains collimator component names, locations, lengths and materials. A `CollimatorDatabase` class constructor accepts this database file as an input argument and collimator information is subsequently added to the existing lattice structure upon calling the `::ConfigureCollimators()` class method. In each component instance a `CollimatorAperture` class is instantiated with information on the jaw gap (using the aperture class discussed in the previous section) and the material.

7.5.2. *Collimator Materials*

To specify collimator properties there is a class `MaterialProperties`. For all materials basic properties such as density are provided. For more specific use-cases, such as resistive wakefields which require to know the conductivity, parameters are not created by default but can be added. The class constructor for `MaterialProperties` takes 8 arguments: A the atomic weight, σ_T, σ_I, and σ_R, the total, inelastic and Rutherford cross sections, dE/dx the energy loss, X_0 the radiation length, ρ the density and Z the atomic number, and these can also be specified by the `::SetMaterialProperties()` method. For properties outside the standard set, the member function `SetExtra()` allows additional parameters to be defined.

For scattering simulations, the cross sections are important. The total and inelastic cross sections, σ_T and σ_I, are specified through `MaterialProperties`. Formulæ do exist which predict them, however they do not always agree with experimental values. As such the user has been given explicit control.

Although some users will want to specify all material properties of a component themselves, most will just want to specify a common material, such as aluminium, copper, or stainless steel. For this purpose, a native `MaterialData` class is provided. Its most common use is as the child class `StandardMaterials` which fills a map whose keys are meaningful character strings for commonly-used materials such as `"Cu"` and `"Fe"`, and whose values are the appropriate `MaterialProperties`.

Material mixtures are specified by reference to entries in the `MaterialData` map. For example

```
StandardMaterialData* matter =
    new StandardMaterialData;
matter->MakeMixture("IT180","W Ni Cu",
   0.95,0.035,0.015,18.06*gram/cc);
```

will create a mixture of Tungsten, Nickel and Copper in the numerical ratios 95:3.5:1.5. (The fractions do not have to add to unity as the numbers are subsequently normalised.) The density has to be specified as it cannot be deduced from the constituent properties. For convenience a `MakeMixtureByWeight` is also provided.

7.6. *Defining the Beam*

A beam can be defined either by specifying individual particle properties or by defining a bunch distribution. In each case an instance of the `ParticleBunch` class is created. For individual particles, the `Particle` class allows the user to specify phase-space information, as defined in Appendix A. Individual particles are added to the bunch via the `::AddParticle(Particle)` method.

To specify particle bunches as a whole, the `ParticleDistributionGenerator` class allows a particle bunch of typical distributions to be constructed. Supported distributions include: uniform, normal/Gaussian, ring, Landau, and Hollow (for HEL simulations). In each case the distribution is determined via random number seeding, outlined in the following section. Also beam parameters must be specified via the `BeamData` class, as it is not inherent. `BeamData` members include: energy `p0`, `charge`, Courant-Snyder parameters `alpha` and `beta`, beam energy and length spreads `sig_dp` and `sig_dp`, and transverse emittances `emit_x` and `emit_y`.

7.6.1. *Random Number Seeding*

When using Merlin++ it is often possible to split the simulation into multiple sub-tasks. For example, a collimation loss map simulation might use 10^8 initial particles, which could be tracked in 100 batches of $1,000,000$ particles (due to the independence between particles). In this case, it is important that the random number streams used are uncorrelated between each batch. This achieved by giving each batch a different initial random seed. It is recommended that a user explicitly defines the seeding of the random number generator, however, if this is not done then a seed is automatically generated using `std::random_device` – considered a high quality random source on most systems. The seed can be set by passing one or more unsigned 32 bit integers to the `RandomNG::init` method. Using multiple values as seed can be useful to avoid statistical correlations between related simulations. For example, when using multiple simulation parameters and multiple batches of particles, then the parameter set number and batch number could both be passed as seed values.

Merlin++ does not aim to guarantee bit for bit reproducibility between multiple runs with the same input. Differences between compilers, compiler options, system libraries and CPU architectures can give small differences to floating point operations. While most modern platforms conform (or can be set to conform) to the IEEE 754 standard for floating-point arithmetic, there are still known discrepancies. Excess precision can occur where intermediate steps of a calculation are not rounded to 64 bit. On modern CPUs, this can happen with fused-multiply-add operations. Minor differences in the implementations of the random number distributions in the C++ standard library are therefore expected.

7.7. *Running a Simulation*

Running a tracking simulation, following configuration of the lattice and beam parameters, takes only a few lines of user code. A `ParticleTracker` class constructor accepts the lattice and particle bunch data as inputs, *e.g.*

```
ParticleTracker
    tracker(myLattice->GetBeamline(), myBeam)
```

A user can then run the tracker simply calling the member function tracker.Run().

8. Summary

Merlin++ continues to be in active development and use. The library contains an extensive set of tools for constructing detailed accelerator models, including standard component types, alignment and geometry information, apertures, and collimator material properties. It contains an accurate high performance tracking integrator, as well as a collection of physics processes needed for advanced accelerator design and optimization. It uses modular object oriented design so that new physics processes can be added as needed by users.

Several features are being considered for future releases. Ion tracking is needed for simulation of accelerators such as RHIC. Variable discretised Truncated Power Series Algebra will enable fast accurate symplectic long-term lifetime studies. An implicit symplectic Runga Kutta integrator would give very accurate tracking, which might be slow but could be used as a benchmark for TPSA accuracy. It will continue to adopt new features in emerging programming standards (such as C++17) to ensure high performance and high quality coding standards. As accelerator hardware become technically more sophisticated to meet future challenges of energy, luminosity, and precision, Merlin++ will continue to provide the simulation facilities needed to ensure successful design and operation of new machines.

9. Acknowledgements

Many people have contributed to the Merlin++ source code and we would like to acknowledge significant contributions from : Adriana Bungau, James Fallon, Maurizio Serluca, Kyrre Ness Sjøbæk and Adina Toader. The current phase of development has been funded by CERN and STFC as part of the HL-LHC project.

Appendix A. The definition of particle parameters

Each particle tracked is specified by 7 numbers, the contents of `PSVector` and also known, through a `typedef`, as `Particle`

1. `x`, the horizontal transverse co-ordinate in metres
2. `xp`, the canonical x momentum, approximately equivalent to the horizontal transverse angle in radians.
3. `y`, the vertical transverse co ordinate in metres
4. `yp`, the canonical y momentum, approximately equivalent to the vertical transverse angle in radians.
5. `ct`, the longitudinal distance from the bunch centre, in metres
6. `dp`, the fractional deviation of the momentum from the nominal value `p0`
7. `id`, which can be used as a unique particle identifier.

On modern x86-64 CPUs we do not find any performance benefit to making the particle storage size a power of 2 for example by adding an additional coordinate or padding.

Merlin does not strictly enforce these units - the user is free to provide processes using different units provided they are consistent within their simulation.

Appendix B. Transfer maps for the Symplectic Integrator

The accelerator Hamiltonian, H, as given in Equation 2.73 of [30] describes the motion of a charged particle in a general electromagnetic field, is written with its six degrees of freedom (transverse positions x and y and momenta p_x and p_y, energy E and longitudinal distance travelled s) converted to canonical co-ordinates using a fixed reference momentum, P_0. This is used to derive the transfer maps.

For a drift

$$x(s) = x + \frac{p_x s}{d}, \tag{B.1}$$

$$p_x(s) = p_x, \tag{B.2}$$

$$y(s) = y + \frac{p_y s}{d}, \tag{B.3}$$

$$p_y(s) = p_y, \tag{B.4}$$

$$z(s) = z + \frac{s}{\beta_0}\Big(1 - \frac{1}{d}\Big) - \frac{\delta s}{d}, \tag{B.5}$$

$$\delta(s) = \delta, \tag{B.6}$$

where

$$d = \sqrt{\Big(\delta + \frac{1}{\beta_0}\Big)^2 - p_x^2 - p_y^2 - \frac{1}{\beta_0^2\gamma_0^2}}. \tag{B.7}$$

and β_0 and γ_0 are the relativistic factors for the reference momentum. Note that due to the square root component, d, the resultant exact drift transfer map is inherently non-linear (this is not the case for `THIN_LENS` and `TRANSPORT` integrators).

The implementation of the Forest sector bend transfer map follows:

$$x(s) = \frac{1}{b_1}\Big(\sqrt{(1+\delta)^2 - p_x(s)^2 - p_y^2} - \rho(\frac{dp_x(s)}{ds} - b_1)\Big) \tag{B.8}$$

$$p_x(s) = p_x \cos\left(\frac{s}{\rho}\right) + \left(\sqrt{(1+\delta)^2 - p_x^2 - p_y^2} - b_1(\rho + x)\right) \sin\left(\frac{s}{\rho}\right) \tag{B.9}$$

$$y(s) = y + \frac{p_y s}{b_1 \rho} + \frac{p_y}{b_1}\left(\sin^{-1}\left(\frac{p_x}{\sqrt{(1+\delta)^2 - p_y^2}}\right) - \sin^{-1}\left(\frac{p_x(s)}{\sqrt{(1+\delta)^2 - p_y^2}}\right)\right) \tag{B.10}$$

$$p_y(s) = p_y \tag{B.11}$$

$$\delta(s) = \delta \tag{B.12}$$

$$z(s) = z + \frac{(1+\delta)s}{b_1 \rho} + \frac{(1+\delta)}{b_1}\left(\sin^{-1}\left(\frac{p_x}{\sqrt{(1+\delta)^2 - p_y^2}}\right) - \sin^{-1}\left(\frac{p_x(s)}{\sqrt{(1+\delta)^2 - p_y^2}}\right)\right). \tag{B.13}$$

In this case, b_1 is a dipole field constant and ρ is the curvature of dipole. The rectangular dipole equivalent, assuming ideal parallel faces, is then derived by taking the limit $\rho \to \infty$.

References

[1] D. A. Edwards, M. J. Syphers, An Introduction to the Physics of High Energy Accelerators, John Wiley and Sons, 1993.

[2] D. Carey, Turtle (trace unlimited rays through lumped elements a computer program for simulating charged particle beam transport systems, Tech. rep., Fermi National Accelerator Laboratory (1978).

[3] K. L. Brown, F. Rothacker, TRANSPORT: a computer program for designing charged particle beam transport systems, Tech. Rep. SLAC-91, SLAC (1983).

[4] K. Makino, M. Berz, Cosy infinity version 9, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 558 (1) (2006) 346 – 350, proceedings of the 8th International Computational Accelerator Physics Conference. doi:10.1016/j.nima.2005.11.109.

[5] R. D. Ruth, A Canonical Integration Technique, IEEE Transactions on Nuclear Science 30 (1983) 2669. doi:10.1109/TNS.1983.4332919.

[6] E. Forest, R. D. Ruth, Fourth-order symplectic integration, Physica D: Nonlinear Phenomena 43 (1) (1990) 105 – 117. doi:10.1016/0167-2789(90)90019-L.

[7] H. Yoshida, Construction of higher order symplectic integrators, Physics Letters A 150 (5) (1990) 262 – 268. doi:10.1016/0375-9601(90)90092-3.

[8] Y. K. Wu, E. Forest, D. S. Robin, Explicit symplectic integrator for s-dependent static magnetic field, Phys. Rev. E 68 (2003) 046502. doi:10.1103/PhysRevE.68.046502.
URL https://link.aps.org/doi/10.1103/PhysRevE.68.046502

[9] J. Laskar, P. Robutel, High order symplectic integrators for perturbed hamiltonian systems, Celestial Mechanics and Dynamical Astronomy 80 (1) (2001) 39–62. doi:10.1023/A:1012098603882.

[10] W. Herr, A MAD-X primer, Tech. Rep. AB-2004-27-ABP, CERN (2004).

[11] D. Sagan, Bmad: A relativistic charged particle simulation library, Nucl. Instrum. Meth. A558 (1) (2006) 356–359, proceedings of the 8th International Computational Accelerator Physics Conference. doi:10.1016/j.nima.2005.11.001.

[12] A. Ferrari, P. Sala, A. Fassò, J. Ranft, FLUKA: a multi-particle transport code, Tech. Rep. CERN-2005-10, CERN (2005).

[13] Merlin++ Developers, Merlin++ (2020). doi:10.5281/zenodo.3700155.
URL https://github.com/Merlin-Collaboration/Merlin

[14] F. Schmidt, Sixtrack: Single particle tracking code treating transverse motion with synchrotron oscillations in a symplectic manner, Tech. Rep. 94-56, CERN (1994).

[15] D. Krücker, F. Poirier, N. Walker, Energy adjustment strategy for dispersion free steering at the ILC using the MERLIN package ILCDFS, Tech. Rep. EUROTeV-Report-2006-106, DESY, Hamburg, Germany (2006).

[16] F. Poirier, D. Krücker, I. Melzer-Pellman, N. Walker, Simulation studies of correlated misalignments in the ILC main linac and the influence of ground motion, Tech. Rep. EUROTeV-Report-2008-017, DESY, Hamburg, Germany (2008).

[17] D. Kruecker, F. Poirier, N. J. Walker, Merlin-based start-to-end simulations of luminosity stability for the ilc, in: 2007 IEEE Particle Accelerator Conference (PAC), 2007, pp. 3277–3279. doi:10.1109/PAC.2007.4440397.

[18] A. Wolski, D. Bates, Spin Tracking Studies for Beam Polarization Preservation in the NLC Main Damping Rings, Linear Collider Collaboration Tech Notes LCC-0155, CBP Tech Note-326, Lawrence Berkeley National Laboratory, Berkeley, California (Jul. 2004). doi:10.2172/889312.

[19] R. J. Barlow, R. B. Appleby, J. Molson, H. Owen, A. Toader, Simulations of the LHC collimation system, Proceedings of IPAC'10 (2010) TUPD061.

[20] J. G. Molson, R. Appleby, R. J. Barlow, M. Serluca, A. Toader, Simulation the LHC collimation system with accelerator physics library merlin, and loss map results, Proceedings of IPAC'12 (2012) MOABC3.

[21] M. Serluca, R. B. Appleby, R. J. Barlow, J. Molson, H. Rafique, A. Toader, Hi-Lumi LHC collimation studies with Merlin code, Proceedings of IPAC'14 (2012) MOPRI077doi:10.18429/JACoW-IPAC2014-MOPRI077.

[22] H. Rafique, R. B. Appleby, R. J. Barlow, R. Bruce, s. Redaelli, S. C. Tygier, Merlin for LHC collimation, Proceedings of IPAC'15 (2014) TUCBC1.doi:10.18429/JACoW-IPAC2015-TUCBC1.

[23] A. Valloni, R. Appleby, R. Bruce, A. Mereghetti, J. G. Molson, E. Quaranta, H. Rafique, S. Redaelli, Merlin cleaning studies with advanced collimator materials for HL-LHC, Proceedings of IPAC'16 (2016) TUCBC1doi:10.18429/JACoW-IPAC2016-WEPMW036.

[24] S. Tygier, R. Appleby, R. Barlow, J. G. Molson, H. Rafique, S. Rowan, Recent development and results with the Merlin tracking code, Proceedings of IPAC'17 (2017) MOPAB013doi:10.18429/JACoW-IPAC2017-MOPAB013.

[25] H. Rafique, J. L. Abelleira, R. Appleby, A. Krainer, A. Langner, Proton Cross-Talk and Losses in the Dispersion Suppressor Regions at the FCC-hh, Proceedings of the 8th Int. Particle Accelerator Conf. IPAC2017 (2017) 1763–1765. doi:10.18429/JACOW-IPAC2017-TUPIK037.

[26] J.-Q. Yang, Y. Zou, J.-Y. Tang, Collimation method studies for next-generation hadron colliders, Physical Review Accelerators and Beams 22 (2) (2019) 023002, publisher: American Physical Society. doi:10.1103/PhysRevAccelBeams.22.023002.

[27] The Eclipse Foundation, Eclipse cdt (c/c++ development tooling) (2020).
URL https://www.eclipse.org/cdt/

[28] R. Brun, F. Rademakers, Root - an object oriented data analysis framework, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81–86.

[29] CMake, Cmake, the cross-platform, open-source build system.
URL https://cmake.org/

[30] A. Wolski, Beam Dynamics in High Energy Particle Accelerators, Imperial College Press, 2014.

[31] A. Wolski, Alternative approach to general coupled linear optics, PRST:AB 9 (2006) 024001. doi:10.1103/PhysRevSTAB.9.024001.

[32] E. Forest, Beam Dynamics: A New Attitude and Framework, Hardwood Academic / CRC Press, 1998.

[33] J. Rosenzweig, L. Serafini, Transverse particle motion in radio-frequency linear accelerators, Phys. Rev. E 49 (1994) 1599–1602. doi:10.1103/PhysRevE.49.1599.

[34] L. Thomas, The kinematics of an electron with an axis, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 3.13 (1927) 1–22. doi:10.1080/14786440108564170.

[35] V. Bargmann, L. Michel, V. Telegdi, Precession of the polarization of particles moving in a homogeneous electromagnetic field, Physical Review Letters 2.10 (1959) 435. doi:10.1103/PhysRevLett.2.435.

[36] H. Burkhardt, Monte carlo generator for synchrotron radiation, LEP Note 632 (1990).

[37] J. Schwinger, On the classical radiation of accelerated electrons, Physical Review 75.12 (1949) 1912–1925. doi:10.1103/PhysRev.75.1912.

[38] T. Böhlen, F. Cerutti, M. Chin, A. Fassò, A. Ferrari, P. Ortega, A. Mairani, P. Sala, G. Smirnov, V. Vlachoudis, The FLUKA code: Developments and challenges for high energy and medical applications, Nuclear Data Sheets 120 (2014) 211–214.

[39] J. Allison, et al., Recent developments in GEANT4, Nucl. Instr. & Meth. A 835 (2016) 186–225. doi:10.1016/j.nima.2016.06.125.

[40] L. Nevay, et al., BDSIM: An accelerator tracking code with particle-matter interactions, Computer Physics Communications 252 (2020) 107200. doi:10.1016/j.cpc.2020.107200.

[41] R. Appleby, R. Barlow, J. Molson, M. Serluca, A. Toader, The practical pomeron for high energy proton collimation, Eur. Phys. J. C 76 (2016) 520. doi:10.1140/epjc/s10052-016-4363-7.

[42] H. Rafique, MERLIN for high luminosity large hadron collider collimation, Ph.D., University of Huddersfield (Apr. 2017).

[43] R. Barlow, A. Bungau, Simulation of high order short range wakefields for particle tracking codes, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 657 (1) (2011) 38 – 44, x-Band Structures, Beam Dynamics and Sources Workshop (XB-10). doi:https://doi.org/10.1016/j.nima.2011.05.013.

[44] A. Toader, R. Barlow, Computation of resistive wakefields, arXiv:901.0860 (2009).

[45] A. Latina, G. Rumolo, D. Schulte, G. A. Blair, S. Malton, J. Smith, R. J. Barlow, A. Bungau, Wakefield models for particle tracking codes, EUROTeV-Report-2007-067 (2007).

[46] D. Schulte, N. Walker, Simulations of the static tuning for the TESLA linear collider, in: Proceedings of the 2003 Particle Accelerator Conference, Vol. 4, IEEE, Portland, OR, USA, 2003, pp. 2736–2738, CERN-AB-2003- 025. doi:10.1109/PAC.2003.1289249.

[47] J. Molson, Proton scattering and collimation for the LHC and LHC luminosity upgrade, PhD, University of Manchester (Dec. 2014).

[48] S. Tygier, R. Appleby, R. Bruce, D. Mirarchi, S. Redaelli, A. Valloni, Performance of the Large Hadron Collider cleaning system during the squeeze: Simulations and measurements, Physical Review Accelerators and Beams 22 (2) (2019) 023001. doi:10.1103/PhysRevAccelBeams.22.023001.

[49] R. C. Martin, Design principles and design patterns (2000).
URL www.objectmentor.com

[50] D. van Heesch, Doxygen.
URL http://www.doxygen.nl/

[51] J. J. Barton, L. R. Nackman, Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples, Addison-Wesley, 1994.

[52] S. van der Walt, S. C. Colbert, G. Varoquaux, The numpy array: A structure for efficient numerical computation, Computing in Science Engineering 13 (2) (2011) 22–30. doi:10.1109/MCSE.2011.37.

[53] M. Matsumoto, T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 3–30. doi:10.1145/272991.272995.

[54] C. Venters, C. Jay, L. Lau, M. Griffiths, V. Holmes, R. Ward, J. Austin, C. Dibsdale, J. Xu, Software sustainability: The modern tower of babel, CEUR Workshop Proceedings 1216 (2014) 7–12.

[55] Software Sustainability Institute, Online sustainability evaluation.
URL https://www.software.ac.uk

[56] S. Rowan, S. Tygier, Y. Cai, C. C. Venters, R. B. Appleby, R. J. Barlow, Sustainability of the Merlin++ particle tracking code, EPJ Web Conferences 214 (2019) 05028. doi:10.1051/epjconf/201921405028.

[57] U. Kunz, Metriculator cdt metric plug-in (2011).
URL https://github.com/ideadapt/metriculator

[58] Valgrind Developer, Valgrind.
URL http://www.valgrind.org/

[59] ArchDia, DV8 Test Suite.
URL https://archdia.com/

[60] Uncrustify Developers, Uncrusifty: Source code beautifier for c, c++, c#, objectivec, d, java, pawn and vala.
URL http://uncrustify.sourceforge.net/

[61] Free Software Foundation, Gnu general public license version 2 (1991).
URL https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html

[62] GitHub, Inc., Github.
URL https://github.com/

[63] D. Schulte, P. Tenenbaum, M. Woodley, N. Walker, A. Wolski, Tests of three linear collider beam dynamics simulation programs, Tech. Rep. SLAC-TN-03-002, LCC-0091, TESLA-2002-08, CLIC-NOTE-513, Stanford Linear Accelerator Center (2002).

[64] J. C. Smith, A. Latina, D. Schulte, F. Poirier, N. Walker, P. Lebrun, K. Ranjan, K. Kubo, P. Tenenbaum, P. Eliasson, Comparison of Tracking Codes for the International Linear Collider, in: Particle Accelerator Conference (PAC 07), IEEE Nucl.Sci.Symp.Conf.Rec., 2007, p. 3020–3022. doi:10.1109/PAC.2007.4440654.

[65] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience., Concurrency - Practice and Experience 17 (2-4) (2005) 323–356. doi:10.1002/cpe.938.

[66] Merlin++ Developers, Merlin++ (2020).
URL http://merlinpp.org

[67] Python Software Foundation, Python.
URL https://www.python.org/