A modular framework for simulations of Ionization Profile Monitors – Implementation and Benchmarking

Dominik Vilsmeier

June 15, 2017

Supervisors:

Prof. Dr. Tilo Wettig
(Universität Regensburg)

Dr. Mariusz Sapinski
(GSI Helmholtz Centre for Heavy Ion Research)

Abstract

Simulations of electron and ion tracking in Ionization Profile Monitors are an important tool for specifying and designing new monitors. They are also essential for understanding the effects related to the ionization process, guiding field non-uniformities and influence of the beam fields which may lead to a distortion of measured beam profiles. Most of these effects cannot be treated analytically and therefore several simulation codes have been developed at different accelerator laboratories during the past years. Those existing codes are often tuned to the specific needs of a laboratory, are not well documented and lack a practical user interface. This work presents a novel, generic simulation tool with focus on the ability to test, maintain and extend the code. A complete documentation as well as facile usage were important aspects too. The application combines the features of existing codes in order to provide a common standard for IPM simulations. Because of its modular structure the application allows for exchanging the computational modules depending on the use case as well as for straightforward extensibility to new use cases. Future intended use cases are for example simulations of Beam Induced Fluorescence monitors based on gas jets or Electron Wire Scanners. The current set of algorithms includes several particle tracking methods (for instance Runge-Kutta 4th order or the Boris algorithm) and several bunch field evaluation algorithms (analytical solutions for specific cases as well as numerical Poisson solvers). The application and all involved methods have been tested and benchmarked against existing results. The code is well documented and includes a graphical user interface. It is publicly available as a git repository and as a Python package.

Contents

List of Figures

1. Introduction

Ionization Profile Monitors (IPM) are used for measuring the transverse profile of particle beams. Those devices make use of the rest gas ionization by the particle beam and guide the ionization products (ions and/or electrons) towards a detector with the help of an external electric field. Figure 1 shows a sketch of an IPM. The main electrodes at the top and bottom and the side electrodes together form the high voltage cage (HV cage). The side electrodes are commonly used for increasing the uniformity of the electric guiding field however designs without side electrodes that have a good field uniformity exist as well [1]. Various acquisition systems are available, figure 1 illustrates the usage of a multi-channel plate (MCP) for amplifying the incoming electron signal, followed by a phosphor screen for converting the electron signal into an optical signal. The produced light is guided by a prism to a camera which records the resulting distribution. The recorded image is a projection of the transverse beam profile for a particular transverse direction. Other possible acquisition systems include arrays of metal strips for detecting the electron signal or pixel detectors which allow for registering a much smaller electron signal [1].

However there are many effects that can influence the quality of measured profiles and thus can make the design and operation of IPMs difficult. Among the effects that directly influence the quality of the measured electron signal the most important ones to mention are [3, 4]:

- *Initial velocities obtained during the ionization process:* Electrons may obtain a substantial kinetic energy during the ionization process and therefore do not move on straight lines towards the detector. Any transverse velocity component introduces a displacement of the measured electrons and therefore a distortion of the measured profiles may become visible.
- *Guiding field non-uniformities:* A uniform guiding field in the relevant detector region is important to ensure that all electrons are likewise guided towards the detector. However

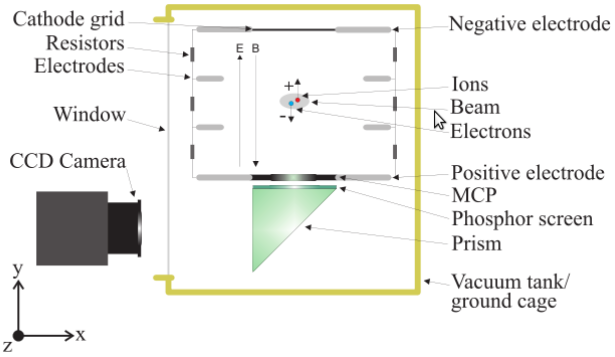

Figure 1: Sketch of an Ionization Profile Monitor [2, H. Refsum] with an optical acquisition system including a phosphor screen, a prism and a camera. A multi-channel plate is used for amplifying the electron signal.

because of the finite size of the electrodes field non-uniformities at the boundaries might play a role and result in a profile distortion.

- *Beam space charge interaction:* The ionization products interact with the electromagnetic field of the particle beam and depending on the beam parameters this interaction can get so strong that the trajectories of particles are significantly distorted resulting in a distortion of the overall profile.

Some of the effects can be compensated by applying an additional magnetic guiding field which is aligned with the electric guiding field. Electrons then spiral around the magnetic field lines and the extent of this gyro motion limits their maximum displacement. However for large beam energies and small beam sizes this compensation might not be sufficient to suppress possible distortions [3].

Because the treatment of those effects is rather complicated and cannot be done analytically it is essential to study them by means of simulations. This is helpful not only for obtaining a deeper understanding of the profile distortions but also for designing devices by verifying that one can expect a good signal quality from a particular set of specifications. For those reasons several simulation codes have been developed at different particle accelerator laboratories during the past years [5–11]. Much interest in further development of IPM simulations has been shown at a dedicated workshop held at the European Center for Nuclear Research (CERN) in 2016 (https://indico.cern.ch/event/491615/) and was reconfirmed at a follow-up workshop held at the GSI Helmholtz Center for Heavy Ion Research (GSI) in 2017 (https://indico.gsi.de/event/IPM17/).

The existing codes are often tuned to the specific needs of the laboratory they have been developed at. Often they are not well documented and do not include a practical user interface which makes it difficult to use or reuse parts of them. The codes often consist of short scripts for Matlab for example and thus the procedures are not easily reusable in other projects. For those reasons the idea of establishing a common simulation tool which combines the features of the existing codes into a single application was born. Such an application should be well documented, tested and benchmarked against existing results as well as measurement data in order to prove the integrity of the implemented methods and to highlight their applicability for different use cases. A clear and modular code structure is highly desirable in order to ensure testability of the single components as well as extensibility to new components and new use cases. The maintainability of the code will also largely benefit from a modular code structure. A graphical user interface is desired in order to facilitate configuring and running simulations as well as to provide simple post-processing tools that allow for quick analyses of the simulated results. Because simulations of other devices such as Beam Induced Fluorescence monitors (BIF) or Electron Wire Scanners involve similar procedures as IPM simulations (most importantly the movement of charged particles in the electromagnetic field of a particle beam along with dedicated methods for generating and detecting those particles) they are foreseen as future use cases as well and thus the design of the application should bear them in mind.

2. Requirements & Use cases

2.1. Use cases

In order to fix the requirements of the application various use cases are discussed below. Defining a clear set of requirements helps to maintain consistency during the development process and avoids unnecessary rework of components or the whole framework. All use cases refer to beam profile monitors which involve the interaction of measured particles with the electromagnetic field of the particle beam and therefore potentially suffer from a displacement which can result in a distortion of the measured profiles.

2.1.1. Profile deformation due to beam space charge

Ionization Profile Monitors make use of the rest gas ionization by the particle beam. By guiding the ionization products towards a detector (multi-channel plate and phosphor screen or pixel detector for example) with the help of an external electric (and magnetic) field one can obtain a projection of the beam's charge distribution for a specific transverse plane. If the beam current is sufficiently high or the beam is sufficiently small then deformation effects due to the interaction of ions or electrons with the beam space charge start to play a role [3, 4]. Instead of traveling on straight lines towards the detector the ionization products suffer from displacements with respect to their initial positions, leading to a deformation of the measured profile. This scenario involves simulating the movement of charged particles (ions or electrons) in the presence of external guiding fields as well as the electromagnetic field of the particle beam. The initial distribution of electrons arises from the interaction of the beam with the rest gas due to ionization. The initial momenta are also obtained during the ionization process. The particles are tracked towards a detector where their final positions and/or momenta are recorded and form the measured profile.

2.1.2. Profile deformation due to guiding field non-uniformities

Guiding field non-uniformities depending on the design of the high voltage cage (HV cage) or the placement of electrodes can have a significant impact on the trajectories of ions or electrons. Those non-uniformities might occur as unwanted by-products in form of fringe fields at the longitudinal boundaries of the HV cage or they might represent a feature of the monitor as described in [12]. With regard to ionization it is important to accurately represent the ionization process in the complete relevant volume of the monitor (which is governed by a pressure bump for example). In addition this scenario involves similar aspects as the profile deformation due to beam space charge (see section 2.1.1).

2.1.3. Simulation of meta-stable excited states and their influence on the measured beam profile (BIF)

Beam Induced Fluorescence monitors (BIF) make use of the excited states of gas atoms or ions which are induced by the interaction with the particle beam. Those excited states decay under the emission of light and the transverse beam profile can be reconstructed provided that the positions of light emission do not change significantly with respect to the corresponding excitation positions. If the particles travel a certain distance until their excited state decays then this displacement might introduce a distortion of the measured profile [13, 14]. This use case aims for investigating the influence of these meta-stable excited states on the displacement of measured particles and therefore on the measured profiles in a BIF monitor. Because the gas atoms are excited by the interaction with the particle beam the distribution of excited atoms follows the distribution of the particles in a bunch. The further such excited atoms travel during the time until the excited state decays the stronger the measured profile will deviate from the actual beam profile. Thus the decay time is deciding for how long the excited particles will be tracked and how large this effect on the measured beam profile is.

2.1.4. Gas jets for IPM and BIF

In case of ultra high vacuums the number of ionized or excited particles might not be sufficient in order to obtain a stable signal for either Ionization Profile Monitors or Beam Induced Fluorescence monitors (the excitation cross section relevant for BIFs is even smaller than the ionization cross section which is relevant for IPMs). Using supersonic gas jets one can create dedicated regions of high gas density which provide good ionization or excitation rates and therefore generate a good signal for the used detector [14]. The gas motion in supersonic gas jets is substantially different from the rest gas motion where the thermal motion dominates. Molecules in the gas jet have one dominant velocity component in the direction of the jet (with a very narrow distribution) and relatively small other components. This additional velocity component is inherited by the ionization products (electrons and ions) and might induce a displacement of the measured particles and therefore a distortion of the resulting profiles. In addition density gradients within the gas jet lead to a signal of spatially varying strength and this effect must be accounted for as well. The goal of this use case is to study the effect of gas jet dynamics on the trajectories of ions and electrons which eventually lead to a distortion of the registered profiles. In case of a BIF monitor gas atoms are excited by the interaction with the beam and tracking must be performed until the excited state decays by emission of light.

2.1.5. Electron background

Electrons may emerge not only from the ionization process involving the interaction of the beam with the rest gas in the vacuum chamber but also from various other sources like the onset of electron clouds or radiation fields due to beam losses [15]. Those electrons are

likewise guided towards the detector and registered as a part of the measured profile. The goal is to study the effect of various background electron distributions on the measured profiles. The background electrons hence need to be described by predefined position and velocity distributions.

2.1.6. Secondary electrons

The ion impact on the electrode or on the ion trap grid (which is installed above the electrode and supplied with a slightly larger voltage) causes secondary electron emission. For an IPM working with electron detection those electrons are likewise guided towards the detector and are registered as a part of the measured profile. The goal of this use case is to study the effect of those secondary electrons on the measured profiles. The final position distribution of ions at the ion trap and the geometry of the ion trap itself defines the initial position distribution of secondary electrons. The momenta of impacting ions must be sufficiently large in order to yield secondary electrons. The relevant threshold is material dependent however for typical IPM specifications it is easily exceeded.

2.1.7. Electron Wire Scanner

For an Electron Wire Scanner an electron beam is aligned perpendicular to the particle beam which causes a deflection of the electrons due to its electric (and magnetic) field. Analyzing the deflection of electrons after they have traversed the beam region one can deduce features of the charge distribution of the beam [16]. A (measured) position and momentum distribution which corresponds to the required electron beam could be provided as an external input resource (emerging from an “electron gun” for example).

2.1.8. Correlations between electron and ion detection

An IPM could detect both electrons and ions emerging from the ionization process with the same beam. Correlations between those measurements allow for conclusions regarding the original beam distribution and could potentially provide deeper insight into the mechanisms of profile distortion. Also one can determine the mode of IPM operation that is suitable for a given beam case.

2.1.9. Simulating multiple beams

Multiple beams (with different particle types) might travel simultaneously through the vacuum chamber as for example in the case of an electron lens or an electron cooler. When measuring the distribution of such beam ensembles the tracked particles are subject to the total electromagnetic field emerging from both beams. Measuring the combined profile of a proton or ion beam which is accompanied by an electron beam (for clearing the beam halo or cooling the proton/ion beam) using a BIF monitor for example the trajectories of excited

ions are influenced by the electromagnetic fields of both beams and will show a different kind of displacement [17]. By using multiple beams one can also emulate the case where consecutive bunches have different parameters as for example for SPS scrubbing runs [18].

2.1.10. Studying trajectories of specific particles

One might want to observe for example only particles which are ionized at the head or the tail of a bunch. Studying trajectories of particles which become trapped inside the bunch (due to the beam field temporarily exceeding the external electric guiding field) is of interest too. By observing the trajectories of single particles one can learn more about the underlying effects that play a role for the displacement of measured particles.

2.2. Requirements

From the above use cases we deduce a set of requirements to which the application shall conform. Those requirements serve as important guidelines during the development process and are chosen such that all discussed use cases can be realized.

1. Each simulation run may involve an arbitrary number of beams (including zero). The combined electromagnetic field of all beams influences the trajectories of tracked particles and therefore the single beam fields cannot be split over multiple runs of the simulation.
2. Each beam consists of one or more identical bunches where two consecutive bunches are separated by an offset greater than zero. Identical means that they have the same shape and parameters, the only difference is their space-time location which leads to different electromagnetic fields measured at a given space-time point in the laboratory frame.
3. Each bunch generates an electric field and is thus assigned a bunch electric field solver. The electric field is solved in the rest frame of a bunch. The electric and magnetic fields as experienced by the tracked particles are obtained by transforming their four vector positions to the rest frame of a bunch, evaluating the electric field in this reference frame and transforming the result back to the laboratory frame which yields the electric and magnetic field as experienced by the particle (using Lorentz transformations). Two identical bunches share the same electric field solver object.
4. Each simulation involves the tracking of exactly one particle type (electrons or ions of a given type). Use cases which involve multiple particle types to be tracked can be realized by running the simulation multiple times (once per particle type) and then merging the results.
5. The tracked particles are non-relativistic and therefore can be described in a single reference frame (the laboratory frame).

6. Each simulation involves exactly one way of generating particles. Use cases which involve particles being created due to different mechanisms can be realized by running the simulation multiple times (once per mechanism) and then merging the results.
7. Each simulation involves exactly one electric guiding field specifier and one magnetic guiding field specifier.
8. The external electric and magnetic guiding fields may vary in time.
9. Each particle is in a defined state at every time during the simulation. Different states are possible, such as "tracked", "detected" or "invalid". A particle's state may be changed by the components of the application.
10. Each simulation takes place in the environment of exactly one "device" which defines the boundaries of the simulation as well as modifies the particles' statuses by deciding when particles are considered to be detected or invalid.
11. The (type of) output of a simulation is configurable however only involves data from the tracked particles. Other data (such as beam field maps for example) shall be generated by other means which are not part of the actual simulation (dedicated scripts can be used for such tasks).
12. The components of the application might depend on each other (for example the particle tracking needs information about the electromagnetic fields). Therefore each component may declare a number of other components as dependencies in order to use attributes or functionality from them.
13. Each component may declare a number of parameters which provide a way of configuring this component through an external configuration source. For each simulation the user must supply such a configuration source which covers all parameters that have been declared by the involved components.
14. The user may specify which particular solution they want to apply to the different part problems of the simulation. For example a user may specify which bunch electric field solver they want to use for computing the beam fields.

Note that the use case "Secondary electrons" (section 2.1.6) can be realized by running two different simulations where the second run requires the output of the first one as an input. That is the first run simulates the ions emerging from the ionization process and estimates their final positions and momenta on the ion trap grid. The second run uses these positions and momenta in order to generate secondary electrons from a dedicated model.

3. Components

The framework covers different components which are discussed in the following sections and which are sketched in figures 2 and 3. The function of the framework is to provide standardized interfaces to the several components as well as corresponding template classes. It should handle all necessary actions that are not directly part of a solution for the different problems that are addressed during a simulation run. The framework includes the necessary background functionality which is completed by the concrete implementations of solutions for the part problems (see section 4).

The application is organized as a Python package and each of the components is represented by a separate sub-package (with exception to the *Output Recorder* component (section 3.8) which is represented by a single module). The framework contains several auxiliary components mostly for storing and modifying particle data and for providing common configuration parameters. Figure 2 shows the structure of the package together with dependencies between the different components and figure 3 shows the dependencies between the core components in detail. Figure 4 shows how those components interact with each other and what tasks they perform related to the simulation loop.

3.1. Model & Manager

Components which represent parts of the application that deal with (physics) problems and which can be addressed in multiple ways feature a concept of *models* and *managers*. A *model* is a specific implementation which addresses the problem in question and which provides the corresponding functionality that is required by a related (template) base class in order to fulfill a common interface. That means a model represents a particular solution for the given problem or aspect (e.g. a numerical Poisson solver is a model which provides a specific way of computing the bunch electric field). A *manager* wraps the corresponding model which is used for a particular simulation run and serves as an entry point for other parts of the framework (hence it can be seen as an adapter for the model which connects it to the rest of the framework). This structure intends to remove any (common) overhead from the specific solutions (the models) and transfers it to the managing component (the manager) which is the same for each model. Such overhead includes data conversions, coordinate transformations or other auxiliary tasks which are not directly related to a specific problem. A model solves the particular problem in a preferably simple and isolated environment and the manager then connects this solution to the application.

3.2. Particle generation

A *Particle Generation Model* represents a way for particles to enter the simulation cycle. For IPM simulations this will most frequently incorporate the ionization process induced by the interaction of the particle beam with the rest gas. Nevertheless implementing other models can be easily realized. For example studying secondary electron emission emerging from

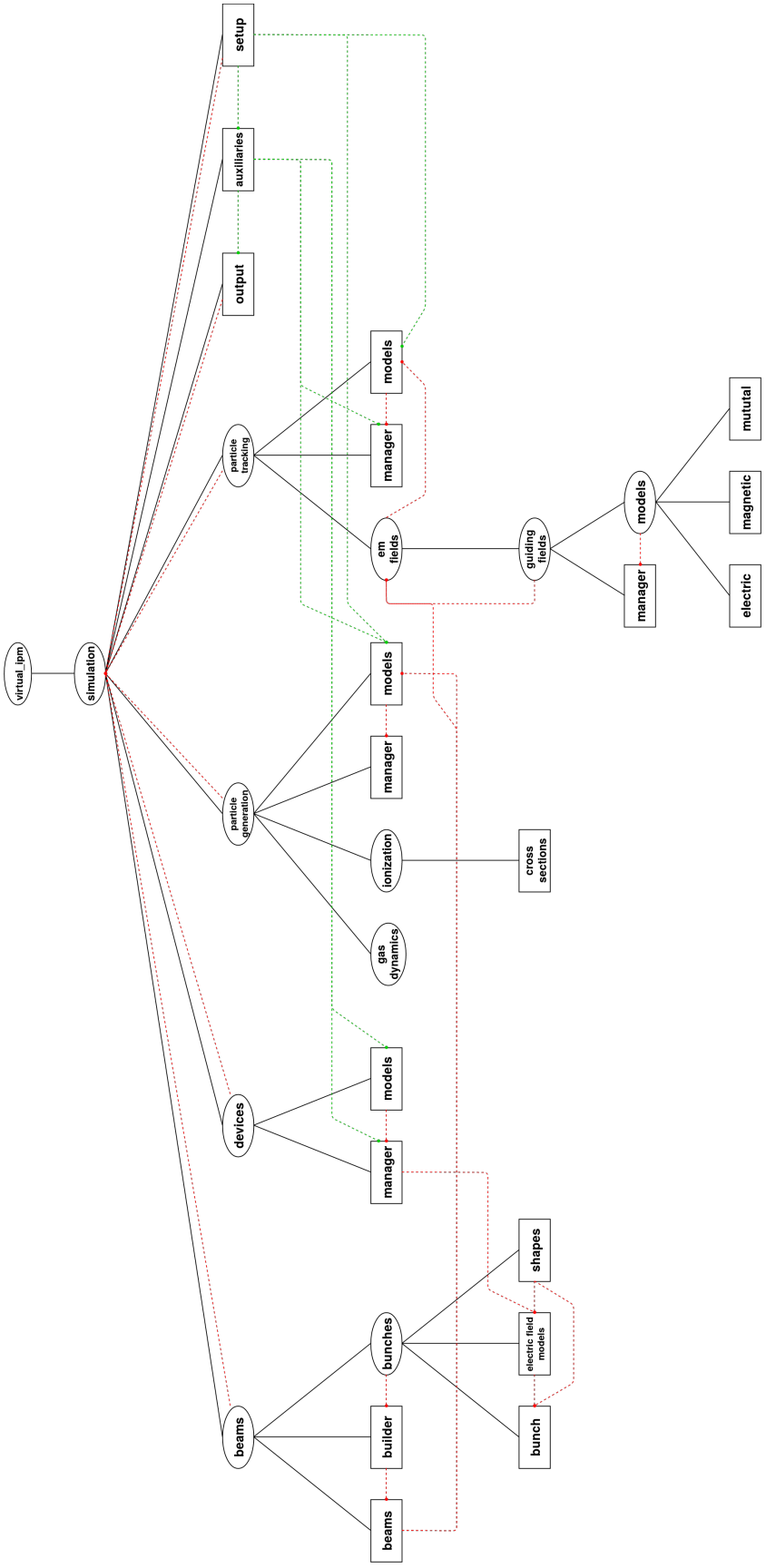

Figure 2: The structure of the software package also indicating the dependencies between the different components. Red links indicate dependencies on core components while green links indicate dependencies on auxiliary components. Ellipses represent sub-packages, rectangles represent modules.

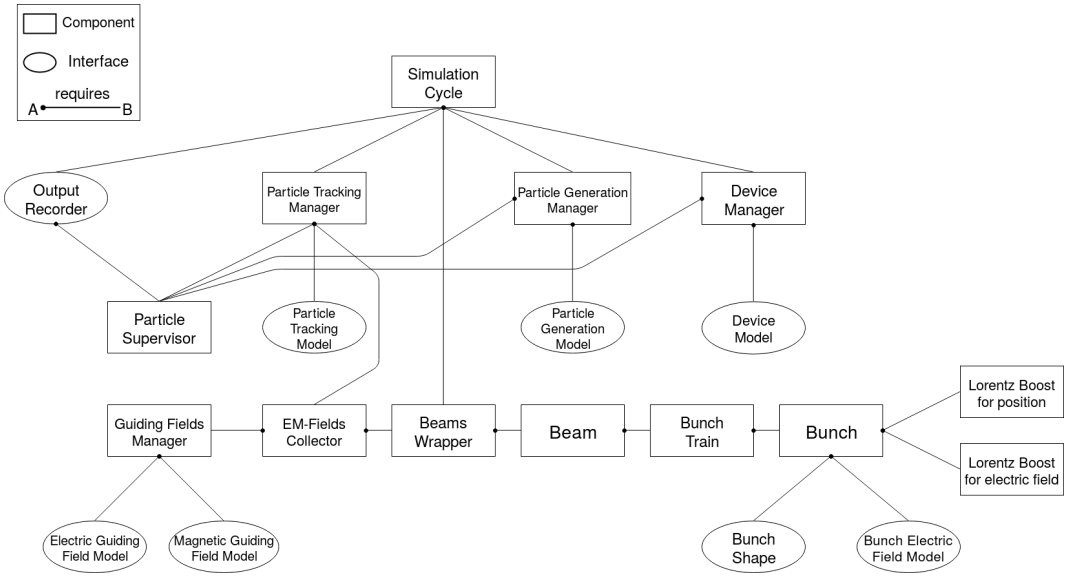

Figure 3: The dependencies between the different core components of the framework.

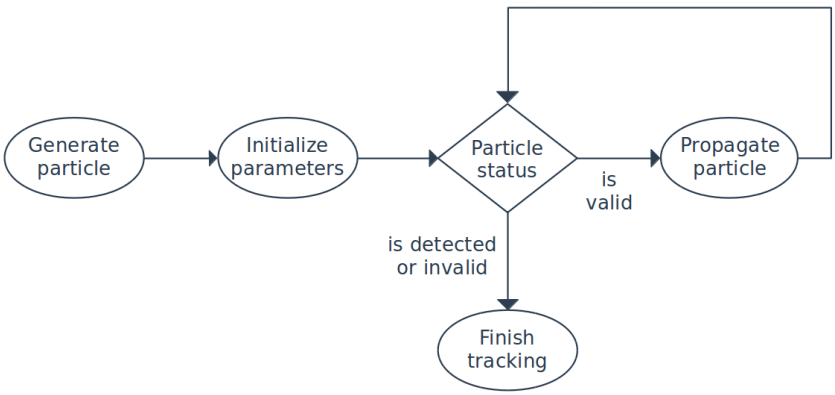

Figure 4: Flowchart showing the actions that take place during the "lifetime" of a particle. Each step is handled by a separate component of the framework.

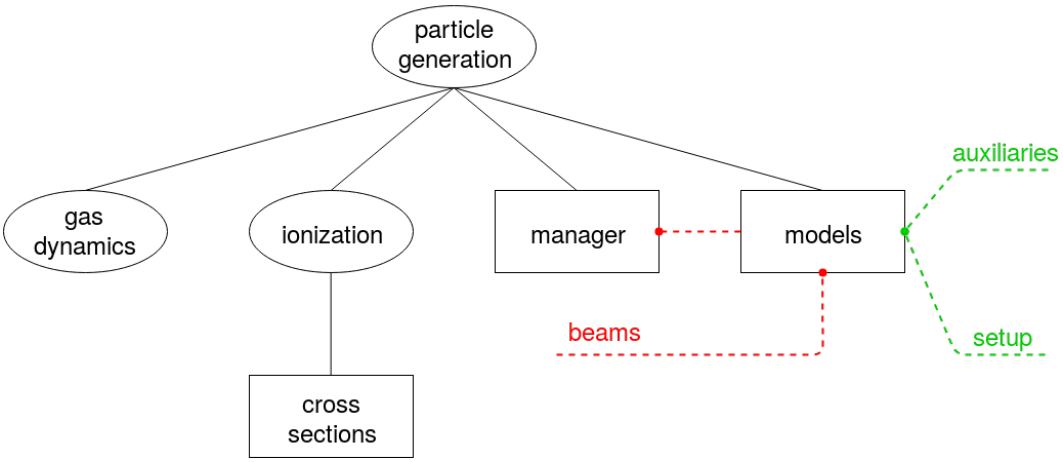

Figure 5: The structure of the particle generation sub-package. Another sub-package for handling tasks related to the gas dynamics is foreseen.

ion impact on detector elements (see section 2.1.6) one would use a model which generates particles based on the output of a previous simulation run during which the ions were tracked towards the detector. In order to simulate an electron gun one could use a corresponding particle generation model which is based on measurements from a real electron gun and which generates electrons according to these distributions. The particle generation model is queried every simulation time step and asked to generate as many particles as appropriate for the given step. As mentioned in section 2.2 each simulation run features exactly one particle generation model. If different particle types are to be analyzed this can be achieved by separate simulation runs. Figure 5 shows the structure of the particle generation sub-package. Besides the common *models* and *manager* modules it contains a sub-package for contents related to *ionization* as for example ionization cross sections. Another sub-package which handles tasks related to *gas dynamics* (becoming important for gas jet simulations for example) is foreseen. Particles are generated by invoking the appropriate methods on the *Particle Supervisor* component (section 3.9.2) which is part of the *auxiliaries* module.

3.2.1. Ionization

For IPM simulations the most common way of generating particles is through ionization. Two aspects are important:

- Initial positions which reflect the bunch's charge distribution.
- Initial momenta emerging from the ionization process.

For the purpose of initial position generation the corresponding bunch shape instance (see section 3.6.2) provides a method for sampling positions according to its (transverse) charge distribution. For the purpose of momentum generation a separate package for ionization cross sections was prepared [19] and the framework exposes several adapters for connecting the

ionization cross section classes of this package to corresponding particle generation models. Such an adapter class provides a method for generating momenta according to the underlying ionization cross section. Different ionization cross sections such as single differential or double differential ionization cross sections are available. In any case they define the energy and the (polar) scattering angle distribution of ionized particles and hence define the resulting momentum distribution.

3.3. Particle tracking

Particle Tracking Models are responsible for propagating particles during the simulation. Each model must provide a method for updating the particles' positions and momenta at a given simulation time step. Propagating particles is an action which needs to be performed per time step and per particle and therefore an efficient method is desirable. On the other hand accuracy of the tracking plays an important role too. This is usually controlled with a time step size Δt by which particles are "pushed" during the current simulation step. A smaller Δt usually implies higher accuracy but also a greater number of updates that need to be performed in total leading to prolonged simulation runs. Thus some kind of trade-off has to be made and an optimum has to be found. Usually it is a good indicator to check a few particle trajectories for different time step sizes and to observe when they converge. Once they do the time step size is reasonably small and can be used to perform a complete simulation run. More about this topic can be found in section 5.1. Figure 6 shows the structure of the particle tracking sub-package. Because the electromagnetic fields determine the particles' trajectories they are wrapped in another sub-package. This sub-package contains the guiding field models and declares a cross-reference to the *beams* sub-package (section 3.6) in order to incorporate the beam fields as well. The relevant particles (those to be tracked) are retrieved from the *Particle Supervisor* component (section 3.9.2) hence the corresponding dependence.

3.4. Devices

On the one hand a *Device Model* defines the spatial boundaries of the simulation (i.e. the relevant simulation region) and on the other hand it decides when the particles' statuses should change because they are detected or invalidated (invalidated means a particle should stop tracking while it was not detected; e.g. because it hit other parts of the chamber than the detector). The boundaries are specified in the laboratory frame (the reference frame in which particle positions are measured) and this boundary information can be reused by bunch electric field models in order to confine the volume in which the electric field has to be precomputed (see section 3.7.1). The task of identifying particles as being detected is a very general one and applies to all the presented use cases (section 2.1). For example when studying a BIF monitor one would use a device model which computes the decay rate of excited states and the corresponding decay probability for each particle. The model then changes the particles' statuses accordingly, based on a (pseudo-) random probabilistic sampling. Once a particle's excited state is decayed the particle is marked "detected" and

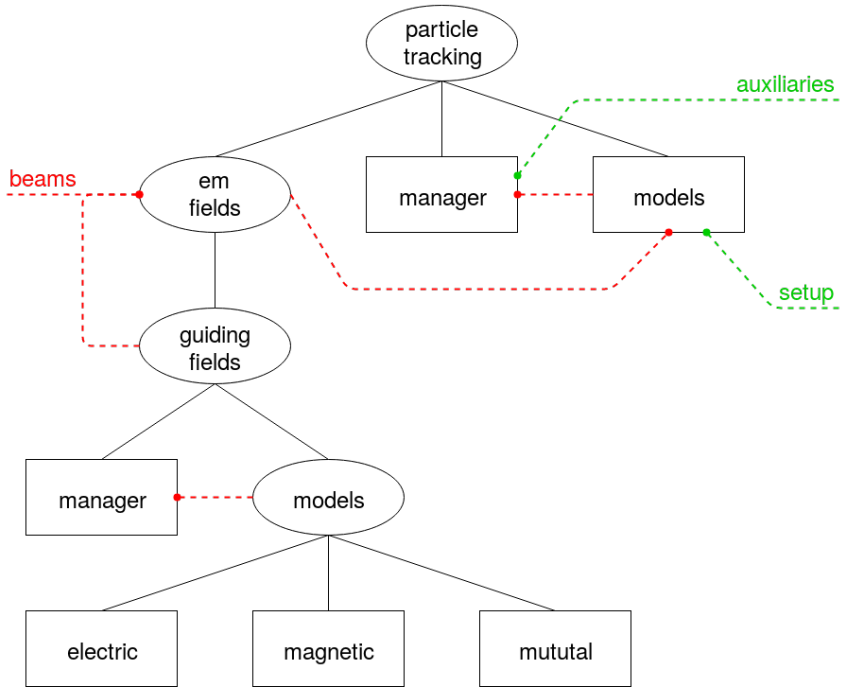

Figure 6: The structure of the particle tracking sub-package.

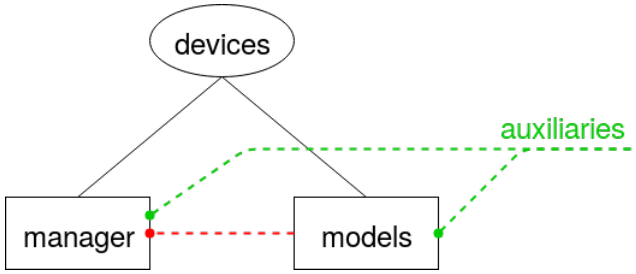

Figure 7: The structure of the devices sub-package.

one can estimate the distance it has traveled and thus compute an influence on the measured beam profile. Figure 7 shows the structure of the *devices* sub-package. In order to check only tracked particles and to change particle statuses a device invokes the corresponding methods of the *Particle Supervisor* component (section 3.9.2).

3.5. Guiding fields

Guiding Field Models describe the external electric and magnetic fields which are applied in order to guide the particles towards the detector. Two kinds of guiding fields are considered, electric and magnetic guiding fields. The guiding fields may vary in time and each simulation run involves exactly one electric and one magnetic guiding field model (if different effects need to be stacked this should be done beforehand, "outside" of the simulation, by use of an appropriate joint model). Because the guiding fields are required for propagating particles

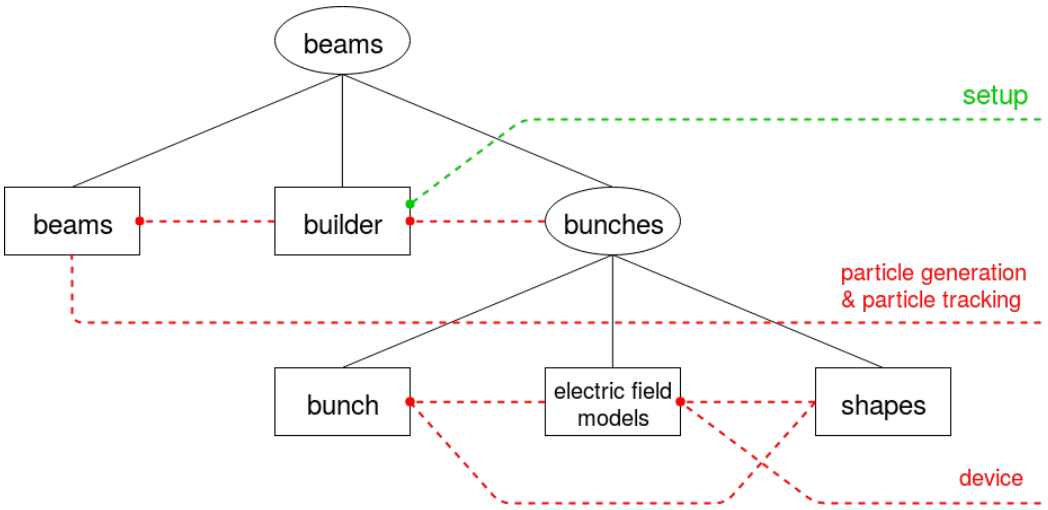

Figure 8: The structure of the beams sub-package.

during the simulation evaluating the fields is an action which takes place per simulation step and per particle and thus should be an efficient procedure. Figure 6 shows the structure of the *guiding fields* sub-package. Because of similarities between the electric and magnetic guiding field models their core functionality has been moved to a separate *mixin* module and the concrete guiding field models are created via (multiple) inheritance from both the mixin model (providing the functionality) and the specific template base class (providing contents relevant for configuration and setup of a simulation run).

3.6. Beams

Each simulation run involves an arbitrary number of beams. Beams take part in processes such as ionization as well as are accompanied by an electromagnetic field which influences the trajectories of particles. Because of this broad range of functionality a separate sub-package *beams* has been created for this component. By convention the bunches of a beam always move in positive z-direction.

3.6.1. Bunch trains

A *Bunch Train* describes the locations of bunches that are part of a beam. Specifically it defines the longitudinal offsets of bunches to the origin of the laboratory frame (the device center) which are used for coordinate transformations from the laboratory frame to a bunch's rest frame.

Linear bunch train For a *Linear Bunch Train* all bunches are placed one behind another and they "move" through the simulation region as the simulation advances (this movement is rather virtual, their longitudinal offsets remains constant, but due to the four vector

transformation of particle positions this is equivalent to the bunches advancing in space). Once a bunch leaves the simulation region it continues to travel in the same direction farther away until the simulation stops. Every two subsequent bunches are separated by the same positive (time-like) spacing. Because the fields of bunches which are farther away may be negligible the user may define a *window width* which determines which bunches in the bunch train should be taken into account for the field evaluation (the window is centered around the device center); this potentially speeds up the simulation as beam field evaluation is computationally rather expensive because it is performed per simulation step and per particle.

Circular bunch train For a *Circular Bunch Train* the bunches are also placed one behind another but with the difference that they are "wrapped around" an (imaginary) boundary outside the simulation region once they reach it during the simulation. This boundary is defined by the number of bunches and their (time-like) spacing. The boundary is placed at $z = N \cdot \Sigma_t/2$ in the laboratory frame where N is the number of bunches and Σ_t is the (time-like) spacing between two subsequent bunches. Because the bunch train is centered around the origin of the laboratory frame this implies that the "rightmost" bunch has a distance of $\Sigma_t/2$ from the boundary. For each simulation time step the actual positions of the bunches are computed by checking their total advancement at the given time step and by wrapping their longitudinal positions around the boundary. If they are wrapped around, they appear on a corresponding left boundary (symmetric with respect to $z = 0$), emulating new bunches coming in the synchrotron.

3.6.2. Bunch shapes

Bunch Shape Models describe the charge distribution of a bunch. They are mainly responsible for two aspects: ionization and electric field computation. With regard to the former aspect each bunch shape model must provide a method for sampling positions of ionized particles according to its transverse charge distribution for a given longitudinal slice (that is the z-coordinate is fixed). If particles need to be generated along the bunch (as it is the case for short bunches) then multiple slices can be used for that purpose. An additional method for doing a complete three dimensional sampling could be implemented as well however for the discussed use cases it is not necessarily required. Also for such a method one must provide a way of confining the sampling with respect to the z-coordinate because for a real device using a pressure bump for example such a confinement takes place (note that this confinement is time dependent in the bunch frame).

For most IPM simulations it will be sufficient to generate particles at $z = 0$ (in the laboratory frame) provided that the guiding fields do not change along the z-axis and that the resulting distribution is integrated along the z-axis. Because the conditions are similar along the z-axis it is not important where particles are created with respect to the laboratory frame. Instead their positions relative to the bunch are important. Therefore it is of importance that the particles are created "along the bunch" (i.e. that their time distribution follows the

longitudinal charge distribution of a bunch).

With regard to electric field computation a bunch shape needs to expose its charge distribution so it can be reused by the bunch electric field model (e.g. a Poisson solver). It might also provide additional optional attributes such as standard deviations of a Gaussian distribution.

3.7. Beam fields

The beam fields are based on the electric field of bunches in their rest frames and are computed as described in this section. Because only non-relativistic particles are considered for the particle tracking the particles' positions can be described in a single reference frame, the laboratory frame (see section 2.2). The positions in a bunch's rest frame can be obtained from the four-vector-positions in the laboratory frame via Lorentz transformation. Furthermore, because of the convention that bunches move along the z-axis, this transformation reduces to a simple Lorentz boost in z-direction (the primed coordinates denote the bunch rest frame):

$$ct' = \gamma(ct - \beta z) \tag{1a}$$
$$x' = x \tag{1b}$$
$$y' = y \tag{1c}$$
$$z' = \gamma(z - \beta ct) \tag{1d}$$

Because each bunch in the bunch train has a different longitudinal offset to the origin of the laboratory frame, we need to replace $ct \to c(t + t_0^i)$ where t_0^i is the time-like longitudinal offset of bunch i. Those offsets are kept constant during the simulation (an exception is the circular bunch train for which the longitudinal offsets are wrapped around a defined window boundary during the simulation) and the appropriate position is obtained from the Lorentz boost. The spatial position in the bunch rest frame is used to evaluate the electric field and the corresponding electromagnetic field tensor is transformed back to the laboratory frame using another Lorentz transformation (note that the bunch magnetic field vanishes in the rest frame of the bunch). Because of the movement along the z-axis this reduces to (the primed coordinates denote the bunch rest frame):

$$E_x = \gamma E'_x \tag{2a}$$
$$E_y = \gamma E'_y \tag{2b}$$
$$E_z = E'_z \tag{2c}$$

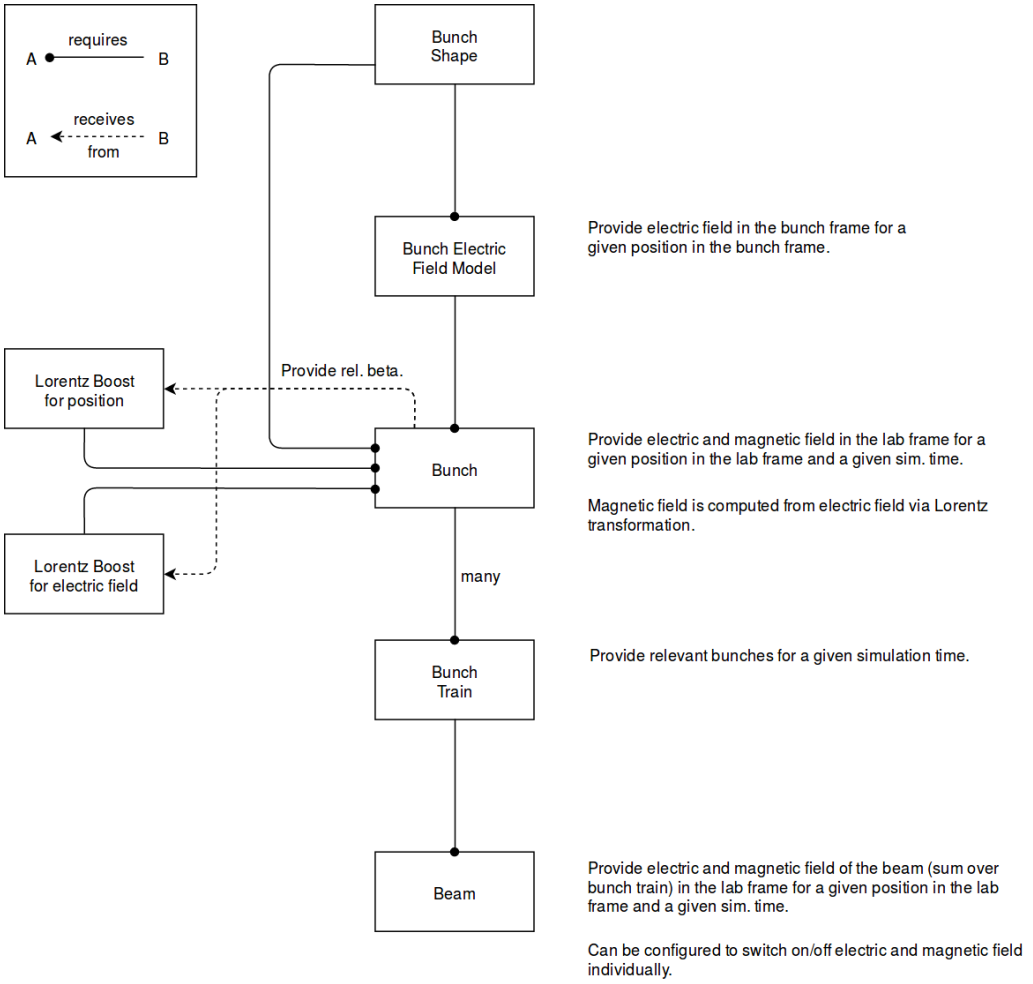

Figure 9: Diagram showing how the electromagnetic field of a beam is obtained.

$$B_x = -\frac{\beta\gamma}{c} E'_y \tag{3a}$$

$$B_y = \frac{\beta\gamma}{c} E'_x \tag{3b}$$

$$B_z = B'_z \tag{3c}$$

3.7.1. Bunch electric field models

Bunch Electric Field Models are responsible for computing the electric field corresponding to a given bunch shape in the rest frame of the bunch. This can be based on analytical solutions for specific cases as well as numerical solutions for arbitrary charge distributions. For that purpose the model can use information from the underlying bunch shape either by accessing certain attributes (e.g. the standard deviations of a Gaussian shape) or by evaluating its

charge distribution from a corresponding method. A bunch electric field model must provide a method for retrieving the electric field evaluated at a given set of positions in the bunch frame. The resulting field of a bunch electric field model is normalized to a bunch charge that equals the elementary charge. The appropriate rescaling with the charge number and bunch population takes place in the corresponding method of a dedicated *Bunch* class which wraps the electric field model and also applies the Lorentz boost.

3.8. Output recorders

Output Recorders are responsible for extracting information about (kinematic) particle data from the simulation, i.e. they are some kind of information sink for particle data (they propagate the desired information to external resources, usually files). One important aspect is that output recorders only deal with particle related information. This means if one wants to retrieve other information such as electric field maps, this information should be obtained by different means (for example dedicated scripts which evaluate the corresponding methods of the related models). Output recorders are subscribed to the status update stream of the *Particle Supervisor* component (section 3.9.2) on which all particle status updates are published and a corresponding callback method is invoked for each status update. An output recorder can react on such status updates by implementing this callback method. In general monitoring particle information involves two aspects:

- Event-based information, such as status changes of particles (e.g. due to ionization or detection).
- Continuous information that must be queried periodically (e.g. particle positions for recording trajectories).

For the second purpose an output recorder's `record` method is called after each iteration of the simulation loop is completed. This method can be overridden in order to periodically request particle data from the particle supervisor component.

3.9. Auxiliaries

3.9.1. Simulation cycle

A *Simulation Cycle* defines when a particular simulation run should terminate. The option which is currently available is a "fixed duration" cycle. That is the user specifies a simulation time and a number of time steps (which together define the time step size Δt) and the cycle runs for the specified number of steps. Therefore it is up to the user to ensure that the specified simulation time is long enough in order to simulate all the effects that are part of the specific case (for example for a bunch with length 10 ns the simulation time should be larger than 10 ns in order to ensure that all particles are created due to ionization; some additional time should be considered for those particles to reach the detector).

Another possible option would be a simulation cycle that runs until all particles have been detected or invalidated. This would allow for using a variable time step size in form of adaptive particle tracking algorithms. The advantage of such tracking algorithms is that they have a better performance because they choose the time step size such that it fits the current field values and field gradients. A fixed time step size Δt must be chosen generally small in order to ensure the field quality at any time during the simulation. An adaptive algorithm could increase Δt whenever the fields or the field gradients are small.

Such a simulation cycle would strongly benefit from a way of generating particles that doesn't involve "slicing" the simulation time as the simulation progress advances. The current implementation uses a "universal" simulation time that advances with the simulation progress (measured in number of elapsed steps). Models are then asked to perform their tasks for the current progress (for example a particle generation model should generate particles according to the beam's charge density at the given universal simulation time). However each particle already maintains a distinct (proper) time in form of the position four-vector and this time is deciding for the Lorentz transformations for example. Therefore the global simulation time could be dropped and the simulation would be described solely by its progress in number of steps. A particle generation model can then create all particles directly at the beginning of the simulation, distributing them properly in space and time. This is advantageous also because if particles are created step-by-step it is not guaranteed that such a simulation cycle won't terminate prematurely (due to no particles being tracked but others still "waiting" to be created).

3.9.2. Particle Supervisor

The *Particle Supervisor* component is responsible for storing and modifying data related to particles. The data of all particles are stored together in numpy arrays [20] and the particle supervisor component provides ways of accessing this data. A *Particle* is a view onto a segment of those arrays and similarly a *Particle Set* is a view on a multitude of such segments (numpy's array indexing is used for that purpose). Those views provide themselves methods for obtaining attributes such as position and momentum corresponding to the represented particles.

3.9.3. Setup

The auxiliary *Setup* component is used to declare universal parameters of the simulation which are the same for each setting, such as simulation time or the number of particles to be simulated. This component can be requested by other components in order to get access to those parameters. The parameters are unique to the *Simulation Cycle* (section 3.9.1) and an additional simulation cycle class would also require an additional corresponding setup class.

3.10. Configuration

Many components need to be configured by means of specifying parameters such as beam energy or bunch population for example. The configuration should be handled automatically and should not be a major concern of components (especially models) of the application. For that reason the required configuration logic was moved to a separate Python package [21]. This package handles all the required tasks related to configuration. In the following, a brief overview of this configuration framework is given.

The framework uses the concept that components of the application *declare* parameters which are then *specified* by the user in a configuration source (usually a file). The configuration framework then handles the transfer of the specified values from the configuration file to the corresponding declaring component. While doing so it checks for any restrictions that were imposed during declaration of a parameter (for example accepting only positive integer values) and applies any necessary transformations (such as unit conversions for example). Figure 10 illustrates the process of configuring components. In the following a brief overview of the available parameter types is given:

- `Bool` - An on/off parameter (switch); *Example: Switch off the electric or magnetic field of a beam.*
- `String` - *Example: Filenames.*
- `Integer` - *Example: The number of bunches in a bunch train.*
- `Number` - A floating point number; *Example: The convergence limit for the Successive Over-Relaxation Poisson bunch field solver.*
- `Vector` - A homogeneous list of an arbitrary parameter type (for example `Vector[Number]`); *Example: The transverse standard deviations of a Gaussian bunch shape.*
- `Duplet` - A vector with two elements.
- `Triplet` - A vector with three elements.
- `Tuple` - A vector with an arbitrary fixed number of elements.
- `PhysicalQuantity` - Needs to be declared with a unit (e.g. `PhysicalQuantity('Energy', unit='eV')`); *Example: Beam energy.*
- `Action` - Must wrap another parameter and specify a custom action. The specified parameter value is transformed using this custom action. This parameter type also allows to declare dependencies on other parameters whose values are then passed as additional arguments to the specified custom action. *Example: Particle types are specified by their charge number and rest energy and the corresponding instance is then created from the* `ParticleType` *class.*

- `Choice` - Needs to be declared with a number of possible options; the specified value must be one of the declared options. *Example: Names of gas types for ionization cross sections.*
- `Group` - Must define a namespace and wrap a number of parameters. The wrapped parameters will appear under the defined namespace in the configuration source. *Example: The tracked particle type is specified by its charge number and rest energy.*
- `ComplementaryGroup` - Must wrap a number of parameters. All but exactly one of the wrapped parameters must be specified by the user and the missing value is computed from the others using previously declared conversion rules. *Example: Simulation time (T), time delta (Δt) and number of time steps (N). Each combination of two of those parameters defines the third one by the relation $T = N \cdot \Delta t$.*
- `SubstitutionGroup` - Must wrap a number of parameters, one of them which is marked "primary". Exactly one of the wrapped parameters must be specified and its value is converted to the corresponding primary parameter's value using previously declared conversion rules. *Example: The user can specify the bunch length either in the laboratory frame or in the rest frame of the bunch. If specified in the laboratory frame then the bunch length is appropriately converted, so the component receives the value in the bunch frame.*

Components also need to declare where their parameters can be found in the configuration file. The framework checks this part of the file and retrieves the corresponding values. In the following an example for a Gaussian bunch shape is given:

```
@parametrize(
    Duplet[PhysicalQuantity]('TransverseSigma', unit='m'),
    PhysicalQuantity('LongitudinalSigma', unit='s')
)
class Gaussian(BunchShape):
    CONFIG_PATH = 'BunchShape/Parameters'
```

`@parametrize` is used to declare the parameters "TransverseSigma" (in units of meters) and "LongitudinalSigma" (in units of seconds; measured bunch length in the laboratory frame). The class attribute `CONFIG_PATH` specifies the location in the configuration file. The corresponding part of an XML configuration file would look like this:

```
<Beam>
    <BunchShape>
        <Model>Gaussian</Model>
        <Parameters>
            <TransverseSigma unit="um">[ 229, 257 ]</TransverseSigma>
            <LongitudinalSigma unit="ns">1.25 / 4</LongitudinalSigma>
        </Parameters>
    </BunchShape>
</Beam>
```

The additional `<Model>` tag specifies that the *Gaussian* bunch shape model should be used.

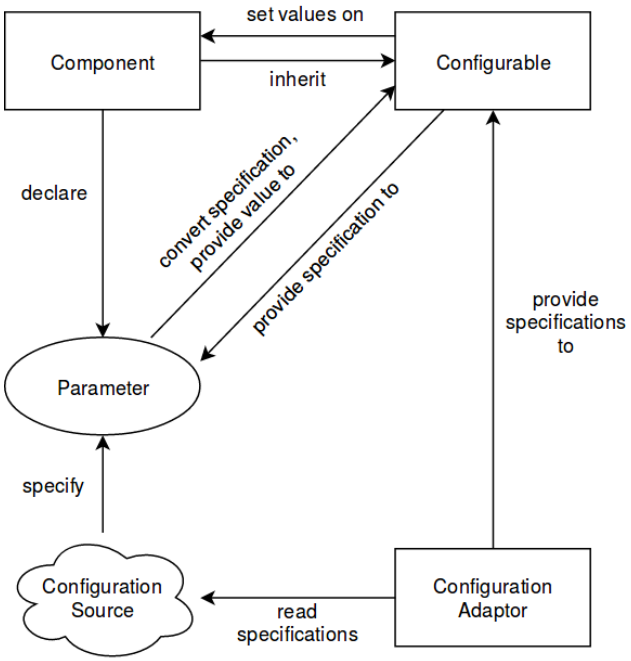

Figure 10: Diagram showing the different aspects of configuring a component as realized by the configuration framework.

Note that the parameters are specified in different units than they were declared; the configuration framework knows the corresponding conversion rules and applies them so that a component always receives the parameters' values as they were declared (that is for the given example in units of meters and seconds, respectively).

3.11. Start & Setup

In order to run a simulation the user must provide a configuration source (usually a configuration file). The configuration source must specify the models which should be used for the different modules (components) as well as all parameters that are declared by the involved models. The recommended way for doing this is by using the Graphical User Interface (GUI) (see section B.3) because it will guide the user through all the different parts of the configuration process and notifies them in case of configuration errors.

The simulation can be started either from the GUI or from the command line (see section B). When started from the GUI a separate simulation thread is created which instantiates the required models and starts the simulation cycle. The simulation thread provides information about the simulation progress and about particles via a publish/subscribe concept (using the ReactiveX library for Python [22]). The simulation GUI subscribes to the corresponding topics and hence provides real-time information about the simulation.

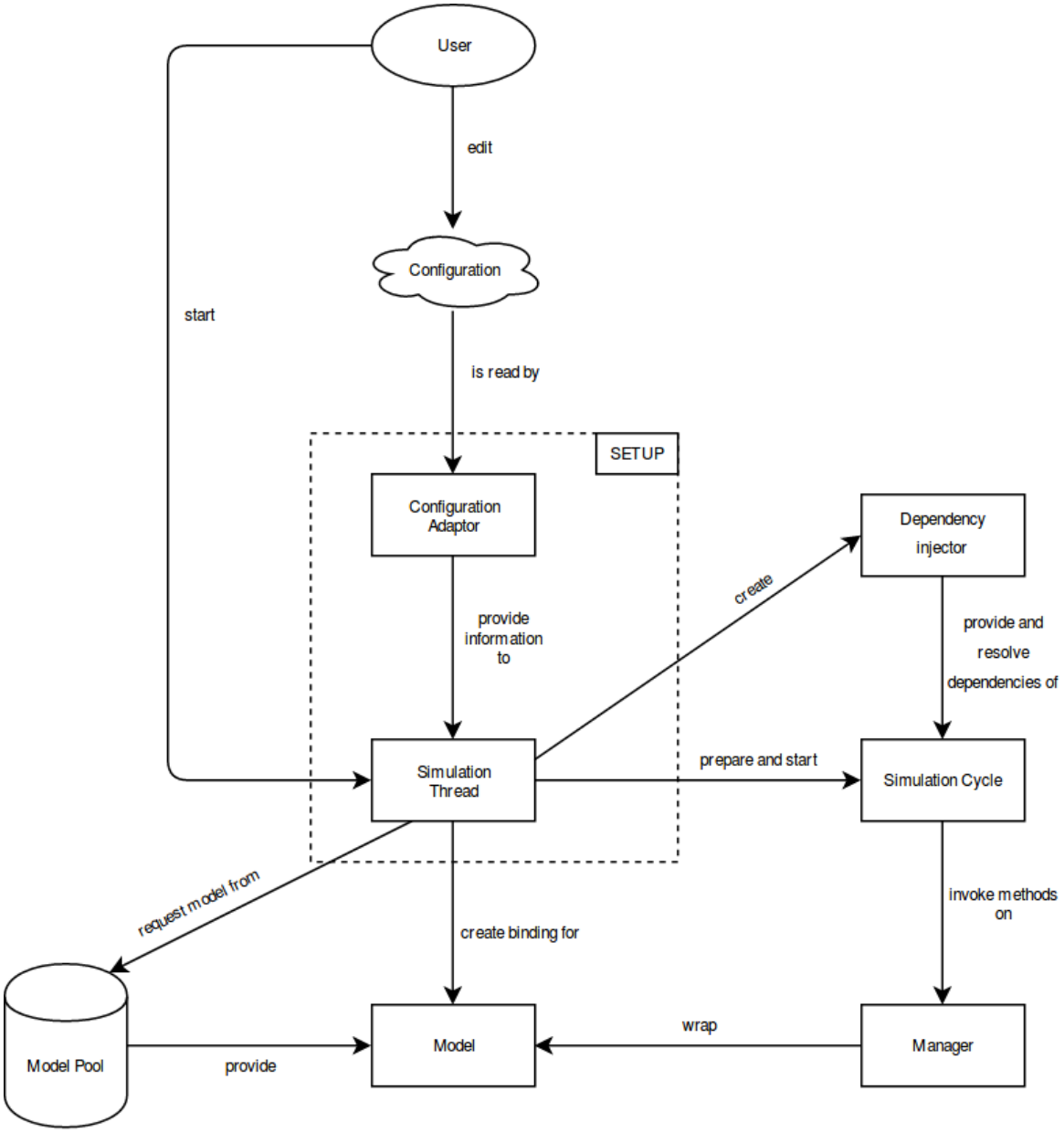

Figure 11: Diagram showing the actions that take place during the start and setup of a simulation run.

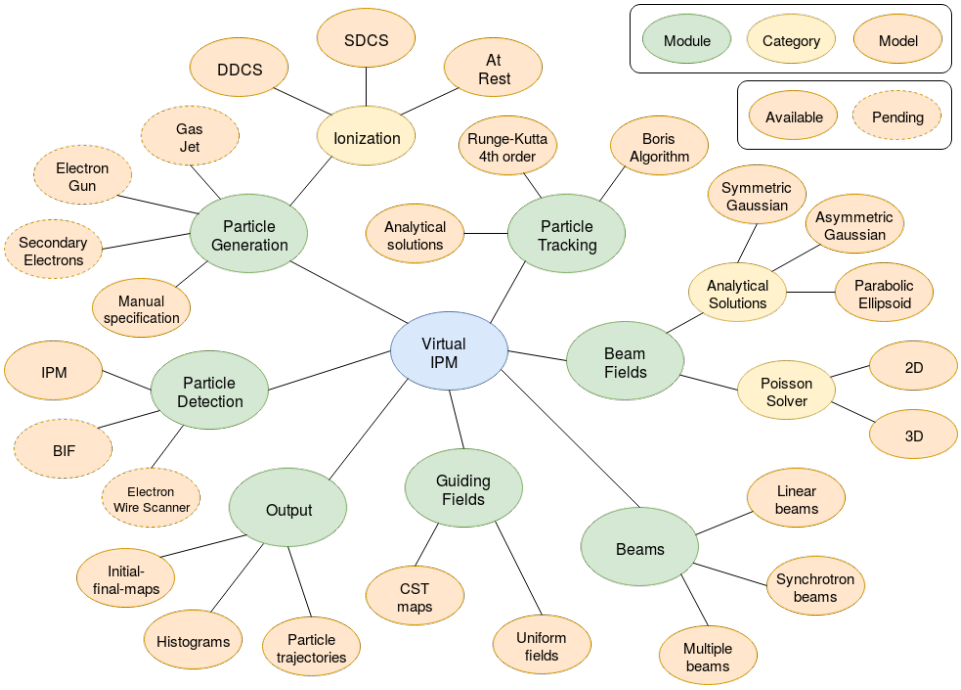

Figure 12: An overview of the available models. Dashed lines indicate future indented models which are not yet implemented.

4. Available models

Several different models have been implemented in order to address the previously mentioned components of the application. Some of these models were inspired or migrated from existing simulation codes for which they already proved successful. The following simulation codes served as a basis for migrating existing models:

- **PyECLOUD-BGI** [7] – An analytical solution for the equations of motion for specific configurations of the electromagnetic fields is used in this code (see section 4.2.3). For the computation of the electric field of the beam this code uses an analytical formula for a two-dimensional elliptical Gaussian charge distribution (see section 4.5.2).
- **GSI-code** [11] – This code uses an analytical solution of Poisson's equation in three dimensions for a specific charge distribution (see section 4.5.3).
- **JPARC-code** [8] – This code uses the Runge-Kutta 4th order algorithm for particle tracking (see section 4.2.1) and a two-dimensional Poisson solver based on the Successive Over-Relaxation method for arbitrary charge distributions (see section 4.5.4).

In the following an overview of the available and intended future models is given. Figure 12 shows these models and to which module they belong.

4.1. Particle Generation

4.1.1. Single particle generation

The *SingleParticle* model allows for placing a single particle in the simulation. The user can specify the time step as well as the initial position and velocity of the generated particle. This is particularly useful for testing configurations by observing real-time trajectories of particles or for studying trajectories of specific particles in general.

4.1.2. Manual generation of particles

Multiple particles can be manually "placed" in the simulation by using the *DirectPlacement* model. The user can specify the simulation steps as well as the initial positions and velocities of the generated particles. This is particularly useful for studying the trajectories of specific particles under certain conditions (e.g. particles which are created at the head or the tail of a bunch). Also if one wants to study the influence of multiple bunches on the trajectories of tracked particles (most likely ions) then it is useful that only the first bunch ionizes all particles (as the situation is similar for following bunches). In order to do so one can run a single bunch simulation, convert the initial parameters from the output to the format of the Direct Placement model and then run a multi-bunch simulation with that parameters as an input.

4.1.3. Ionize particles at a fixed z-position

If the guiding fields of a device do not change along the z-axis and the resulting signal is integrated along this axis (while the profile along, for example, the x-axis is measured) then it is sufficient to generate all particles at a fixed z-position. Because the conditions are similar along the z-axis and for the beam fields only the relative positions of the tracked particles to a bunch are important we only need to make sure that the particles are properly distributed "along the beam". This is achieved by fixing the z-position and distributing the particles in the time domain as the simulation advances, following the charge distribution of the beam for fixed $z = 0$ for example (note that a bunch's time- and z-coordinate are coupled by $z = \beta c(t + t_0)$, where t_0 is the bunch's initial (time-like) offset to the device center). The momenta of ionized particles can be obtained with different methods. For example generating particles at rest or using ionization cross sections for estimating the momenta is available. Figure 13 sketches how particles are distributed during the simulation for this approach.

This functionality has been implemented in the *FixedZZeroMomentum*, *FixedZVoitkivDDCS* and *FixedZSimpleDDCS* models which feature different methods for sampling the momentum.

4.1.4. Ionization cross sections

Ionization cross sections describe the energy and the scattering angle of products of ionization processes. Different types of cross sections - single and double differential ionization cross

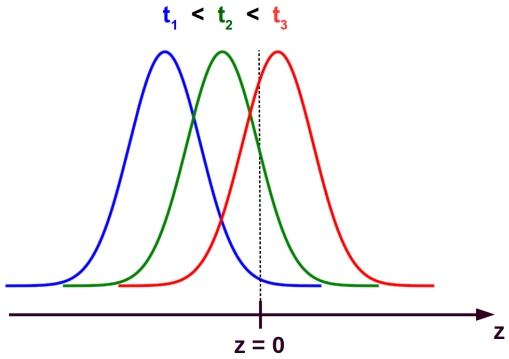

Figure 13: Sketch of how particles are generated at $z = 0$. The different colored distributions indicate the same bunch at different times during the simulation. The particles are sampled according to the distribution which is found at $z = 0$ for the given time step.

sections - have been implemented for the application.

Double differential cross section (DDCS) The ionization cross section derived by Voitkiv et. al. [23] is applicable to the case of highly relativistic incident particles as well as projectiles with high charge numbers ((a) $v \lessapprox Z \ll v^2\gamma, 1 \ll v < c$ or (b) $Z \ll v, v_0 \ll v < c$, in atomic units, Z being the charge number of the projectile). It is derived in a quantum mechanical approach by solving the non-relativistic Schroedinger equation with the help of Coulomb continuum wave functions for the case of hydrogen. An extension to helium is realized by the introduction of an effective charge as seen by the electrons in the 1s-shell of the helium atom. Therefore the applicability of this method to multi-shell atoms is questionable. The double differential cross section is given by the following expression [23, eq. (38)]:

$$\begin{aligned}\frac{d^2\sigma_{He}^{(+1)}}{dEd\Omega} = 2 \cdot 2^8 \frac{Z^2}{v^2 Z_t^4} \frac{1}{\left(1+\frac{2E}{Z_t^2}\right)^5} \frac{\exp\left(-\frac{4\arctan\sqrt{\frac{2E}{Z_t^2}}}{\sqrt{\frac{2E}{Z_t^2}}}\right)}{1-\exp\left(-\frac{2\pi}{\sqrt{\frac{2E}{Z_t^2}}}\right)} \\ \times \left[\sin^2\theta \cdot \ln\eta_{He} + \frac{\cos^2\theta}{\gamma^2} - 0.5\sin^2\theta \right. \\ \left. + \frac{8\sqrt{2E}}{v}\cos\theta \cdot \left[1-\frac{v^2}{2c^2}\right]\sin^2\theta \ln\eta_{He} + \frac{2ZZ_t}{v^2\gamma^2}\cos\theta\ln^2\eta_{He}\right]\end{aligned} \tag{4}$$

where Z is the charge number of the projectile, a_0 is the Bohr radius, Z_t is the effective charge of helium (determined to a value of 1.5), η_{He} is a coefficient specific to helium, γ is the relativistic factor and v is the velocity of the projectile. The cross section is given in atomic units. Figure 14 shows a plot of the cross section for 6.5 TeV incident protons on a helium target. The plot shows that most ionized electrons have small kinetic energies ($E < 10\,\text{eV}$) as it is typically the case for ionization by relativistic projectiles.

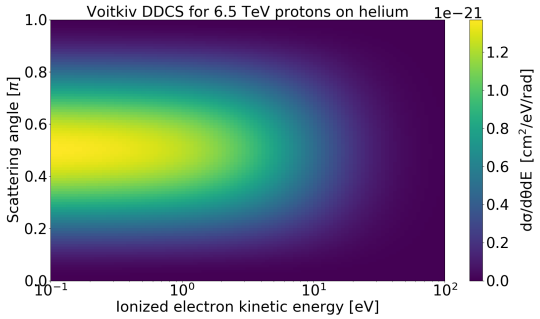

Figure 14: Double differential cross section for 6.5 TeV incident protons on a helium target.

Single differential cross sections (SDCS) For ionization by highly relativistic projectiles the interaction time is very short and the situation can be treated as a two-body problem; the electron in the vicinity of the nucleus receives energy almost instantly in form of a short electromagnetic pulse originating from the rapidly varying electromagnetic field of the projectile (similar to photoionization). For those cases the main contribution to the ionized electron kinetic energy comes from the kinetic energy distribution of the bound electron. Corresponding single differential cross sections (SDCS) have a typical shape which consists of a low-energy plateau part, representing the Compton profile of the target atom, followed by a slope which depends on the various shell contributions (see figure 15). This plateau-slope shape can be parametrized by the four parameters σ_P (the plateau value of the SDCS), σ_S (the SDCS value at the end of the slope), E_P (the energy where the plateau ends and the slope starts) and E_S (the energy where the slope ends); the plateau is assumed to start at $E = 0$. Note that this plateau-slope shape occurs in a log-log form and therefore the underlying distribution for the slope part is of the form

$$\frac{d\sigma}{dE}^{slope} = \frac{\sigma_P}{E_P^a} \cdot E^a \quad , \quad a = \frac{\log\left(\sigma_S/\sigma_P\right)}{\log\left(E_S/E_P\right)} \tag{5}$$

The corresponding scattering angle distributions for ionization by highly relativistic projectiles typically have a Gaussian shape that is centered around $\theta = \pi/2$. For projectile velocities close to the speed of light no "dragging" of the electron occurs during the ionization process (that is the electron would be accelerated in the direction of movement of the projectile) and therefore the skewness of the scattering angle distribution can be assumed to be zero. Figure 16 shows the corresponding single differential cross section for 6.5 TeV incident protons on a helium target.

4.1.5. Electron gun (pending)

For studying electron wire scanners an electron gun needs to be emulated. A corresponding model could take a measured energy/emission-angle distribution as an input (from a data file or an analytical/numerical model for example) and use this distribution to generate electrons during the simulation.

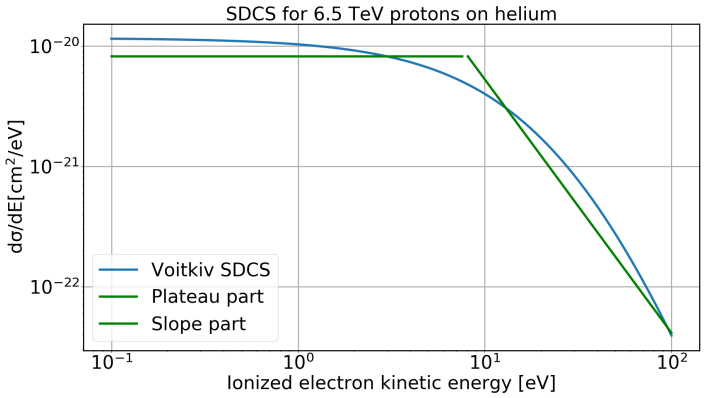

Figure 15: Single differential cross sections for 6.5 TeV incident protons on a helium target. The blue curve shows the ionization cross section as derived by Voitkiv et. al. [23, eq. (39)]. The green curve represents a parametrization to reflect the typical plateau and slope part of such ionization cross sections. The values $\sigma_P = 8.2 \times 10^{-21}, \sigma_S = 4.0 \times 10^{-23}, E_P = 8\,\text{eV}, E_S = 100\,\text{eV}$ have been used for the parametrization.

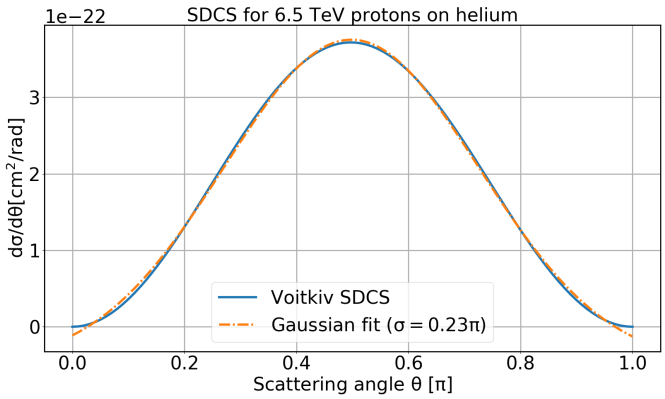

Figure 16: Single differential cross sections for 6.5 TeV incident protons on a helium target. The blue curve shows the ionization cross section as derived by Voitkiv et. al. [23, eq. (40)]. The orange curve represents a Gaussian fit to reflect the typical shape of the scattering angle ionization cross section.

4.1.6. Secondary electrons (pending)

Secondary electrons are generated from ions hitting the ion trap grid. In a first simulation run one would simulate the movement of ions towards the grid and record their impact positions and momenta. This output from a previous simulation run can then be used as the input to a secondary electron generation model which creates electrons according to a model which describes the interaction of impacting ions with the material of the ion trap.

4.1.7. Gas jet (pending)

A gas jet model incorporates ionization via the particle beam. Depending on the distribution of gas density in the jet the actual number of particles generated by the corresponding bunch shape model must be adjusted (rescaled) accordingly. The momenta of ionized particles are adjusted by supplying the velocity component of the gas jet itself. That is the ionized particles (ions or electrons) retain the velocity of the molecules in the gas jet.

4.2. Particle tracking

4.2.1. Runge-Kutta 4th order

The Runge-Kutta method is a general method for solving ordinary differential equations of the form

$$\frac{d}{dt}y(t) = f\left(y(t), t\right) \tag{6}$$

by bringing them into a linear form. Runge-Kutta methods exist of various orders which characterize the number of intermediate evaluations of $f(y,t)$ which are used to determine the "next" value of $y(t)$. A starting point $y_0 \equiv y(t_0)$ needs to be supplied. From this starting point on the next value of $y(t)$ is estimated by a linear update (linear in t). This is achieved by discretizing the time domain. The following considerations refer to the 4th order Runge-Kutta method. For a finite update of length Δt the updated value of $y(t)$ is computed as:

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{7}$$

where the k_i are evaluations of $f(y,t)$ between $[y_n, y_{n+1}]$:

$$k_1 = f\left(y_n, \quad t_n\right) \tag{8a}$$
$$k_2 = f\left(y_n + \frac{\Delta t}{2}k_1, \quad t_n + \frac{\Delta t}{2}\right) \tag{8b}$$
$$k_3 = f\left(y_n + \frac{\Delta t}{2}k_2, \quad t_n + \frac{\Delta t}{2}\right) \tag{8c}$$
$$k_4 = f\left(y_n + \Delta t k_3, \quad t_n + \Delta t\right) \tag{8d}$$

In the case of non-relativistic equations of motion this becomes:

$$\frac{d}{dt}\vec{x} = \vec{v} \tag{9a}$$
$$\frac{d}{dt}\vec{v} = \frac{q}{m}\left(\vec{E} + \vec{v} \times \vec{B}\right) \tag{9b}$$

We solve both equations "in lockstep" using the updated values of velocity for an intermediary position update. That is the coefficients are computed as:

$$k_{x1} = \vec{v}_n \tag{10a}$$
$$k_{v1} = \frac{q}{m}\left(\vec{E}_n + \vec{v}_n \times \vec{B}_n\right) \tag{10b}$$
$$k_{x2} = \vec{v}_n + \frac{\Delta t}{2}k_{v1} \tag{10c}$$
$$k_{v2} = \frac{q}{m}\left(\vec{E}_n + \left(\vec{v}_n + \frac{\Delta t}{2}k_{v1}\right) \times \vec{B}_n\right) \tag{10d}$$
$$\ldots \quad \text{and so on}$$

Note that the field is only retrieved at the beginning of an update and then used for each intermediary step. For that reason Δt has to be chosen sufficiently small if the electric or magnetic field has large field gradients (which invalidate the uniformity assumption during an update). This consideration is especially important for scenarios where beam space charge effects play a major role. However also for large guiding field non-uniformities this effect might become relevant. A possible improvement would be to retrieve field also at the intermediary positions $x_n + k_{xi}h$ where h is either $h = \Delta t/2$ or $h = \Delta t$ depending on the intermediary step and to use this field for further computations of k_{vi} (and k_{xi} therefore).

4.2.2. Boris algorithm

The Boris algorithm is a particle pusher algorithm which is widely used in plasma simulations. It performs a velocity update by using a transformation which separates the effects of the electric and magnetic field [24]:

$$\vec{v}^1 \equiv \vec{v}_n + \frac{q\vec{E}}{m}\frac{\Delta t}{2} \tag{11a}$$

$$\vec{v}^2 \equiv \vec{v}_n - \frac{q\vec{E}}{m}\frac{\Delta t}{2} \tag{11b}$$

This transformation eliminates the electric field from the equations of motion and results in a pure (energy conserving) rotation due to the magnetic field:

$$\vec{v}^2 = \vec{v}^1 + \left[\frac{q}{2m}\left(\vec{v}^1 + \vec{v}^2\right) \times \vec{B}\right] \Delta t \tag{12}$$

The updated velocity can therefore be obtained by applying the above transformation, updating the transformed velocity and transforming the result back. The updated velocity is then used to update the particle's position. This is an explicit method in the sense that is only uses the fields from the beginning of an update rather than incorporating the fields from an updated position as well (as it is the case for implicit schemes like for example the *Backward Euler Method*). Because the fields at the beginning of an update are known this reduces the amount of (computational) effort for performing an update. The implementation of the Boris algorithm includes a time-shift by $-\Delta t/2$ of momentum with respect to position (momentum is shifted backwards in time by $\Delta t/2$; see figure 17). This has the positive effect that for each velocity update the field at the corresponding intermediary position is used: for $\vec{v}_n \to \vec{v}_{n+1}$ the corresponding position at the beginning of the update is $\vec{x}_{n+1/2}$ because of the shift by $-\Delta t/2$. Because the updated position is obtained from the updated velocity both updates use the field at this intermediary position. This has the advantage that the approximation of the fields during a velocity update improves from $\mathcal{O}(1)$ (constant field) to $\mathcal{O}(x)$ (linear field). Note that this shift by $-\Delta t/2$ is not required but convenient for cases which involve large field gradients. Thus we can obtain the initial shift by either applying the method itself on the non-shifted parameters or using another method that doesn't incorporate such a time-like shift. Shifting position and velocity could be applied to other methods as well, provided that they use the updated velocity for performing a position update ("leapfrog" scheme).

4.2.3. Analytical solutions

An analytical solution can be written for the special case of constant and uniform B_x, B_y, E_x, E_y and $B_z \equiv E_z \equiv 0$ (both conditions must apply). For the solution of the equations of motion and the resulting position and velocity updates see [7, p.34-38]. Note

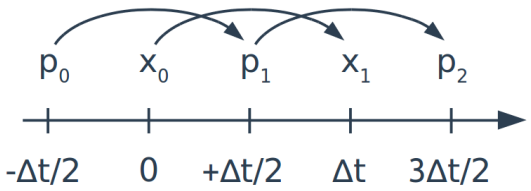

Figure 17: Sketch of momentum being shifted by $-\Delta/2$ against the position for the Boris algorithm.

that the above requirement implies that the electric and magnetic fields are assumed to be constant during an update and therefore the time step size Δt must be chosen small enough so that the quality of this assumption holds for the (spatial) distance of the update. This becomes particularly important for large field gradients for which one needs to make sure that the time step size is small enough (see section 5.1.3). This method has been implemented in the *RadialFields* model along with an optimized version for the case $B_x = 0$ in the *RadialFieldsBx0* model.

4.3. Devices

4.3.1. Ionization Profile Monitor

The *BasicIPM* model implements the functionality of an Ionization Profile Monitor. It checks the positions of particles and estimates whether they have reached the detector level (the multichannel-plate for a real IPM for example) or if they hit a surrounding boundary of the chamber. Particles that have reached the detector are marked as "detected" and their final parameters are stored. Particles store their position and velocity as well as their previous position and velocity and therefore the final positions at the detector are linearly interpolated by the *InterpolatingIPM* model. If a higher order of interpolation is desired then complete particle trajectories can be computed (see section 4.6.2) and used later for higher order interpolation.

4.3.2. Beam Induced Fluorescence monitor (pending)

A device model representing a Beam Induced Fluorescence monitor would incorporate the decay curve of excited states of the involved atoms or ions. For each tracked particle the model would estimate - based on a stochastic process - whether the particle remains in an excited state or decays. Whenever an excited state decays the particle is marked "detected" and its final position is stored. By doing so one can simulate how far particles travel from the position of their excitation to the position where their excited state decays under the emission of light which is then registered by the BIF monitor. This allows for estimations of profile distortion due to the displacement of excited particles.

4.4. Guiding fields

4.4.1. Uniform fields

Often the assumption of a uniform guiding field is sufficient. The *UniformElectricField* and *UniformMagneticField* models provide a guiding field which is constant and has the same value at every point in space. If the design of the high voltage cage provides a field distribution which deviates only outside the relevant detector region then this model is a sufficient choice.

4.4.2. Field maps

Two- or three-dimensional field maps obtained from external programs such as CST Studio [25] or Poisson Superfish [26] are commonly used. Those external programs estimate the electric and magnetic fields based on numerical solutions of Maxwell's equations. The *CSVAdaptor2D* and *CSVAdaptor3D* models support electric and magnetic field maps presented in a standardized comma-separated values (csv) format. The actual field values are obtained via interpolation from the field values at the given grid points. A tool for converting files in CST format to the standardized csv format exists as part of the application (see section B.5).

4.5. Bunch electric field models

4.5.1. Symmetric Gaussian (analytical, 2D)

The transverse part of a symmetric Gaussian charge distribution is described by the following formula:

$$\rho(x,y) = \frac{Q}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{13}$$

Solving Poisson's equation in two dimensions using polar coordinates for the above charge density one obtains for the electric field:

$$E_r = \frac{Q}{2\pi\epsilon_0}\frac{1}{r}\left(1-\exp\left(-\frac{r^2}{2\sigma^2}\right)\right) \quad , \quad r^2 \equiv x^2+y^2 \tag{14}$$

In order to take the longitudinal dimension into account the resulting field is rescaled with the fraction of the longitudinal charge density:

$$E_r \to E_r \cdot \rho_z(z) = E_r \cdot \frac{1}{\sqrt{2\pi\sigma_z^2}} \exp\left(-\frac{z^2}{2\sigma_z^2}\right) \tag{15}$$

Note that $\int_{-\infty}^{+\infty} \rho_z(z)dz = 1$ and therefore $\rho_z(z)$ represents the fraction of the total longitudinal charge density at position z. The field does not include a longitudinal component ($E_z \equiv 0$) and therefore this model is only suitable for long bunches ($\sigma_z \gg \sigma$). The transverse field components are computed as:

$$E_x = E_r \cos\varphi \tag{16a}$$

$$E_y = E_r \sin\varphi \tag{16b}$$

where φ denotes the azimuthal angle.

The corresponding implementation is the *AnalyticalSolutionForRadiallySymmetricGaussianBunch* model.

4.5.2. Asymmetric Gaussian (analytical, 2D)

For an asymmetric Gaussian charge distribution ($\sigma_x \neq \sigma_y$) with the charge density described by

$$\rho(x,y) = \frac{Q}{2\pi\sigma_x\sigma_y} \exp\left(-\frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\right) \tag{17}$$

one can solve Poisson's equation using the complementary complex error function [27]. The field components are evaluated as:

$$E_x = \frac{Q}{2\epsilon_0\sqrt{2\pi(\sigma_x^2-\sigma_y^2)}}\Im\left[\mathrm{erfcx}\left(\frac{x+iy}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right) - \exp\left(-\frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\right)\cdot\mathrm{erfcx}\left(\frac{x\frac{\sigma_y}{\sigma_x}+iy\frac{\sigma_x}{\sigma_y}}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right)\right] \tag{18}$$

$$E_y = \frac{Q}{2\epsilon_0\sqrt{2\pi(\sigma_x^2-\sigma_y^2)}}\Re\left[\mathrm{erfcx}\left(\frac{x+iy}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right) - \exp\left(-\frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\right)\cdot\mathrm{erfcx}\left(\frac{x\frac{\sigma_y}{\sigma_x}+iy\frac{\sigma_x}{\sigma_y}}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right)\right] \tag{19}$$

where $\Im$ denotes the imaginary part and $\Re$ denotes the real part; erfcx is the complementary complex error function [28]. In the above formula $\sigma_x > \sigma_y$ is assumed. For $\sigma_y > \sigma_x$ the result can be obtained likewise by exchanging $x \leftrightarrow y$, $\sigma_x \leftrightarrow \sigma_y$ and $E_x \leftrightarrow E_y$. In order to

take the longitudinal dimension into account the resulting field is rescaled with the fraction of the longitudinal charge density:

$$E_{x,y} \to E_{x,y} \cdot \rho_z(z) = E_{x,y} \cdot \frac{1}{\sqrt{2\pi\sigma_z^2}} \exp\left(-\frac{z^2}{2\sigma_z^2}\right) \tag{20}$$

Note that $\int_{-\infty}^{+\infty} \rho_z(z)dz = 1$ and therefore $\rho_z(z)$ represents the fraction of the total longitudinal charge density at position z. The field does not include a longitudinal component ($E_z \equiv 0$) and therefore this model is only suitable for long bunches ($\sigma_z \gg \sigma_x$ and $\sigma_z \gg \sigma_y$).

The corresponding implementation is the *BassettiErskine* model.

4.5.3. Parabolic ellipsoid (analytical, 3D)

For a charge density that is confined by a rotationally symmetric ellipsoid (symmetric with respect to the z-axis) defined by the equation

$$\frac{r^2}{b^2} + \frac{z^2}{a^2} \leq 1 \tag{21}$$

and which is parabolic inside as well as zero outside of the ellipsoid, as described by

$$\rho(r,z) = \begin{cases} 15Q/8\pi ab^2 \cdot \left(1 - r^2/b^2 - z^2/a^2\right) & , r^2/b^2 + z^2/a^2 \leq 1 \\ 0 & , r^2/b^2 + z^2/a^2 > 1 \end{cases} \tag{22}$$

one can find an analytical solution for Poisson's equation in three dimensions by using a transformation to elliptical (curvilinear) coordinates [29]. In the formulas above a and b denote the semi-major and the semi-minor axis of the ellipsoid ($a > b$), respectively. Because the equations for the field components are rather lengthy we don't quote them here but refer to [29] instead.

It has been found that for very long bunches ($a/b \gtrapprox 10,000$) the field computation suffers from numerical uncertainties and therefore yields invalid results. By using a library for arbitrary floating point precision [30] and observing the field to converge when going to higher precision (see figure 27) it has been verified that this is solely an issue with numerical precision (rather than an algorithmic error). The error occurs during the computation of various coefficients that are later used for the field evaluation and a solution is being investigated.

The corresponding implementation is the *ElectricFieldOfParabolicEllipsoid* model.

4.5.4. Poisson solver based on Successive Over-Relaxation (numerical, 2D)

Solving Poisson's equation for an arbitrary charge distribution in two dimensions can be achieved by discretizing the spatial domain and applying the *Successive Over-Relaxation*

(SOR) method [31]. The potential is determined during an iterative process with a convergence limit ϵ. The spatial domain is represented by a $N_x \times N_y$ grid and the charge distribution is evaluated at each of the grid points. An initial guess for the potential (e.g. zero) is made for each of the lattice sites as well. The grid contains two extra rows and columns at the boundaries to represent the Dirichlet boundary conditions. The potential is fixed on these boundaries prior to the start of the method (e.g. zero for conducting boundaries). The method is then applied by performing sweeps over the lattice, updating one lattice site at a time, until convergence is reached. The potential update $\phi_{x,y}^{i+1}$ at lattice site (x, y) is computed as:

$$\begin{aligned}\phi_{x,y}^{i+1} =&(1-\xi)\phi_{x,y}^{i} \\ &+ \frac{\xi}{2(d_x^2+d_y^2)}\left(\frac{\rho_{x,y}}{\epsilon_0}d_x^2 d_y^2 + (\phi_{x+1,y}^{i}+\phi_{x-1,y}^{i+1})d_y^2 + (\phi_{x,y+1}^{i}+\phi_{x,y-1}^{i+1})d_x^2\right)\end{aligned} \tag{23}$$

where $1 < \xi < 2$ is the *over-relaxation parameter*, $\rho_{x,y}$ is the charge density at lattice site (x, y), d_x, d_y are the lattice spacings in x- and y-direction, respectively, and ϵ_0 is the vacuum permittivity. The convergence factor is calculated as

$$\tilde{\epsilon}^{i+1} = \max\left(\frac{\left|\phi_{x,y}^{i+1} - \phi_{x,y}^{i}\right|}{\max\left(\left|\phi_{x,y}^{i+1}\right|\right)}\right) \tag{24}$$

and convergence is reached when $\tilde{\epsilon}^{i+1} < \epsilon$.

4.5.5. Poisson solver based on Finite Elements (numerical, 3D)

The *Finite Element Method (FEM)* is a general method for solving partial differential equations under certain boundary conditions. The idea behind this method is to divide the domain of interest into multiple smaller domains (finite elements) which are governed by simpler equations and to connect those domains to a larger set of equations by respecting the continuity at their boundaries. Solving this larger set of linear equations yields an approximate solutions at each of the grid points that confine the finite elements. The application uses the FEM framework FEniCS [32, 33] to solve Poisson's equation for the electric field of a bunch. As of now a regular three dimensional box mesh is used for discretizing the spatial domain. The source term (the charge density) is evaluated on each of the grid points. In order to solve the differential equation a function space must be specified. This is done by using polynomials of a variable degree (Lagrange polynomials of degree 1 are used by default). Once a solution for the potential has been found the corresponding electric field can be computed by taking the (finite difference) derivative of the potential on the grid and projecting it back onto the specified function space. For example for polynomials of degree 1 the resulting field would be approximated by a stepwise linear function. Taking the derivative and back-projecting is conveniently handled by the framework. Using higher order polynomials requires the solver to store more coefficients for the polynomials (one per

degree) and therefore uses more memory; the same holds for a larger number of grid points. More about performance and memory consumption can be found in section 5.4.2. Because the charge density and the gradient of the charge density is not the same everywhere in the simulation region the method could be improved by using a variable mesh that fits the given charge density (instead of a regular mesh). The variable mesh could be adapted such that the gradient of the charge density between two neighboring lattice sites is (approximately) constant over the whole mesh.

4.6. Output recorders

4.6.1. Mapping of initial to final particle attributes

Often one wants to compare initial profiles with the corresponding final profiles. Having a per-particle mapping of initial to final attributes is even more convenient because it allows for studying the change of those attributes for specific particles or specific initial or final conditions (see for example figure 32). Such functionality is available through the *BasicRecorder* class. This class also provides several parameters for tuning which attributes should be saved. The content of a corresponding output file can be visualized from the graphical user interface (see section B.4.1).

4.6.2. Studying trajectories

Studying trajectories of specific particles is possible with the *TrajectoryRecorder* class. The user can specify a number of particles via their IDs (particle IDs start at zero and are incremented by one for each particle generated) and the trajectories of those particles will be saved to dedicated files. Figure 41 shows a particle trajectory that is obtained in real-time. By using the Trajectory Recorder class those values are similarly saved to a file.

4.6.3. Beam profiles in XML format

The *XMLProfileRecorder* class records initial and final x-positions of particles and - when the simulation finished - generates corresponding histograms from these positions. The corresponding profiles are saved together with the complete configuration information to a common file, following a standardized XML format [34]. This format can be read and visualized by a dedicated data analysis application [35].

5. Benchmarking

In order to check the accuracy, applicability and efficiency of the various models several tests and benchmarking cases have been performed. The purpose of *testing* is to verify that a certain method works as expected by checking general properties or by comparing the results against (analytically) known results. The purpose of *benchmarking* is to compare the results of a model or an ensemble of models against either data from other simulation tools or measurement data.

For the following tests and benchmarkings three different settings were chosen: one for the Large Hadron Collider (LHC) as well as for the Proton Synchrotron (PS), both operated at CERN, and one case for the heavy ion synchrotron SIS-18 at the GSI Helmholtz Centre for Heavy Ion Research. The two former cases involve significant profile distortions due to beam space charge interaction and are therefore interesting to study. However only simulation data is available for those case. The latter case refers to experimental data which was taken in June 2016 at SIS-18. The measurements were done with both electron and ion detection in order to have a reliable reference. Table 1 shows the parameters of the three benchmark cases. Note that for the PS case there is no external magnetic field and therefore the kind of distortion will be very different from the one of the LHC case. The SIS-18 IPMs are operated without magnetic field however beam space charge interaction is expected to be smaller for this case (note the small bunch population of 2.0×10^7).

	LHC case	**PS case**[1]	**SIS-18 case**
Particle type	Protons	Protons	$^{124}Xe^{43+}$
Energy/u	6.5 TeV	25 GeV	600 MeV
Bunch population	1.3×10^{11}	1.33×10^{11}	2.0×10^7
Bunch length (4σ)	1.25 ns	3.0 ns	44 ns
Bunch width (1σ)	229 µm	3.7 mm	7.81 mm
Bunch height (1σ)	257 µm	1.4 mm	3.26 mm
Electrode distance	85 mm	70 mm	180 mm
Applied voltage	4 kV	3 kV and 20 kV	8 kV
Magnetic field	0.2 T	0 T	0 T

Table 1: Parameters for the studied benchmark cases.

5.1. Particle tracking

5.1.1. Gyro motion

A common scenario will be the tracking of particles in a homogeneous magnetic field which is aligned with an electric field as for the LHC case. The electric field moves the particles towards the detector and the magnetic field confines their maximal displacement by their gyroradius. For particle tracking models it is therefore important to accurately reproduce

[1] The real PS IPMs use a magnetic field of 0.2 T. The presented use case demonstrates the necessity of using a magnetic field in order to prevent profile distortions.

this gyromotion in order to allow for reliable results. Therefore we test the accuracy of tracking algorithms for the case of $\vec{E} = (0, E, 0)$, $\vec{B} = (0, B, 0)$ and without beam fields. For this setup we can obtain the analytical solutions for the equations of motions as:

$$x(t) = x_0 + \frac{v_x}{\omega}\sin(\omega t) + \frac{v_z}{\omega}(\cos(\omega t) - 1) \tag{25a}$$

$$y(t) = y_0 + v_y t + \frac{q}{2m}Et^2 \tag{25b}$$

$$z(t) = z_0 + \frac{v_z}{\omega}\sin(\omega t) + \frac{v_x}{\omega}(1 - \cos(\omega t)) \tag{25c}$$

with the cyclotron frequency $\omega = qB/m$. The following parameters were chosen: $B = 200\,\mathrm{mT}$, $E = 4\,\mathrm{kV}/85\,\mathrm{mm}$, $v_x = 0.035\,c_0, v_y = v_z = 0$. The resulting gyroradius is $R = 300\,\mu\mathrm{m}$. A large initial velocity (gyroradius) has been chosen as an extreme case of electrons being strongly kicked by the bunch electric field [3, Fig. 11]. The absolute position error increases with the magnitude of the motion and this scenario is intended to represent an upper limit for real cases. The particles were tracked for 30 gyration periods (4.47 ns). Figure 18 shows the results of the test. The PyECLOUD-BGI tracking algorithm is an analytical solution for the presented case and therefore performs likewise good for any of the step sizes. Both the Runge-Kutta and the Boris algorithm show an increasing accuracy for decreasing time step size Δt whereas the Runge-Kutta seems to increase faster. This is explained by the fact that the Runge-Kutta algorithm uses four intermediate steps for an update by Δt and therefore uses a time step size which is effectively smaller (however not simply $\Delta t/4$ due to the kind of intermediary "stepping", see section 4.2.1). The deviation in y-direction where only the electric field is acting and causing an acceleration has been found to be negligible, equally for all algorithms as shown in figure 19. The slight increase in position error for larger number of time steps is explained by numerical errors being added up more often and therefore leading to a greater deviation in the end. Because for a pure gyro motion the gyro momentum and the corresponding energy is conserved this is a suitable test quantity as well. Figure 19 shows the energy conservation for the different models. The Boris algorithm shows very good energy conservation for all time step sizes because - other than the Runge-Kutta - the Boris algorithm is phase-space volume conserving (although it is not symplectic) and therefore conserves energy (i.e. it involves a bound on the associated error in energy) [36]. The energy conservation of the Runge-Kutta, however, becomes reasonably small already for larger time step sizes.

5.1.2. E × B-drift

The beam electric field acting perpendicular to the external magnetic field induces a E×B-drift and therefore tracking algorithms must reproduce this behavior correctly too. For constant electric and magnetic fields $\vec{E} = (E, 0, 0)$, $\vec{B} = (0, B, 0)$ the solution for the equations of

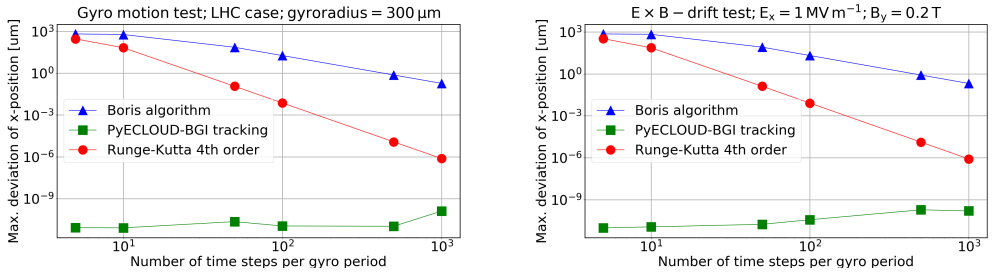

Figure 18: Results for the accuracy tests for tracking algorithms. Left: Pure gyro motion. Right: E × B-drift.

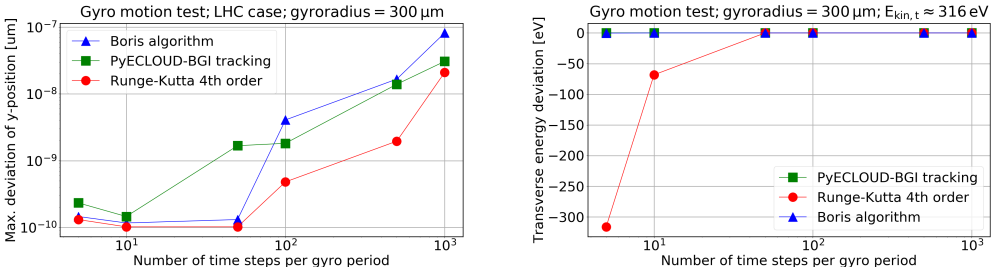

Figure 19: Results for the gyro motion test case. Left: y-deviation due to tracking. Right: Energy conservation of tracking models.

motion is obtained as:

$$x(t) = x_0 - \frac{v_z}{\omega} + \frac{E}{\omega B} - \left(\frac{E}{\omega B} - \frac{v_z}{\omega}\right)\cos(\omega t) + \frac{v_x}{\omega}\sin(\omega t) \tag{26a}$$

$$y(t) = y_0 + v_y t \tag{26b}$$

$$z(t) = z_0 + \frac{v_x}{\omega} + \frac{E}{B}t - \left(\frac{E}{\omega B} - \frac{v_z}{\omega}\right)\sin(\omega t) - \frac{v_x}{\omega}\cos(\omega t) \tag{26c}$$

with the cyclotron frequency $\omega = qB/m$. The term $E/B\,t$ in $z(t)$ describes the drift. The following parameters where chosen: $B = 200\,\mathrm{mT}$, $E = 1\,\mathrm{MV\,m^{-1}}$, $v_x = 0.035\,c_0, v_y = v_z = 0$. The corresponding gyroradius is $R = 300\,\mu\mathrm{m}$. A large initial velocity has been chosen as an extreme case of electrons being strongly kicked by the bunch electric field [3, Fig. 11]. The absolute position error increases with the magnitude of the motion and this case is intended to represent an upper limit for real cases. The same holds for the large value of the electric field. Actual field values can reach as high near the center of the bunch (see figure 25). Figure 20 shows the resulting trajectories and figure 18 shows the results of the test. Similar to the pure gyro motion test the PyECLOUD-BGI tracking algorithm represents an analytical solution for the presented case and thus performs very good. Again the Runge-Kutta's accuracy increases faster for similar arguments as for the pure gyro motion case (section 5.1.1).

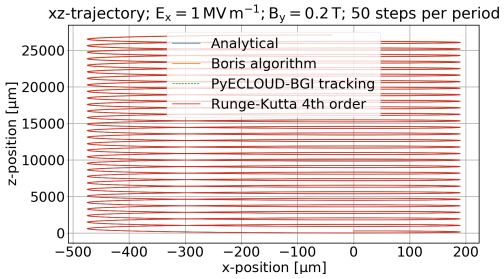

Figure 20: Trajectories for the E × B-drift test case.

5.1.3. Trajectories with beam fields

LHC case While the previous tests were performed without influence of the beam fields we now discuss the results for the real cases including those fields. A test particle is placed at $x = y = z = 0$ with an initial energy of 1 eV (this is a typical value obtained from a corresponding double differential cross section, see figure 14) and the momentum vector pointing in x-direction. The bunch has an offset of $z_0 = -4\sigma_z$ to the device center. Figure 21 shows the trajectories for the LHC case for different numbers of time steps per gyro period (that is for different Δt). The left plot shows a significant enlargement of the gyroradius as well as a significant E × B-drift for both the Runge-Kutta and the PyECLOUD-BGI tracking algorithm compared to the Boris algorithm. Increasing the number of time steps (that is decreasing the time step size Δt) shows that the trajectories converge towards the result of the Boris algorithm (the right plot). For 50 time steps per gyro period ($\Delta t \approx 3.57\,\mathrm{ps}$) both the Runge-Kutta and the PyECLOUD-BGI tracking algorithm show significant distortions.

This can be explained by the fact that both algorithms evaluate the electromagnetic fields at the beginning of an update and then use these values for the complete "push" (which implies the assumption that the fields are constant during the temporal (and spatial) length of the push). For large fields and large field gradients this uniformity assumption is invalidated (see section 5.2) and thus leads to invalid trajectories.

Figure 22 shows the electric field of a bunch and also visualizes how the field is evaluated during a particle update. Because of the steep field gradient the field is overestimated for particles that move towards the bunch center ($x = 0$). From figure 22 one can deduce a field gradient of approximately $9.25\,\mathrm{kV\,m^{-1}\,\mu m^{-1}}$ for the linear region $-0.3\,\mathrm{mm} < x < 0.3\,\mathrm{mm}$. Considering the kinetic energy of 1 eV and the time step size of 3.57 ps this leads to a distance of approximately 2.1 µm that the particle travels during an update. This means a change of $19.5\,\mathrm{kV\,m^{-1}}$ in the electric field during the update. So the field actually varies significantly on the distance that the particle moves during an update however only the initial value is taken into account. This overestimation leads to an acceleration of the particle which is too large. Taking into account the acceleration of the electron $a = qE/m$ this adds an extra $6.1 \times 10^3\,\mathrm{m\,s^{-1}}$ to the velocity which is 1% of the particle's initial velocity. The same holds for particles moving away from the bunch center. The (absolute value of the) field strength actually increases during the push however only the initial value is used leading to

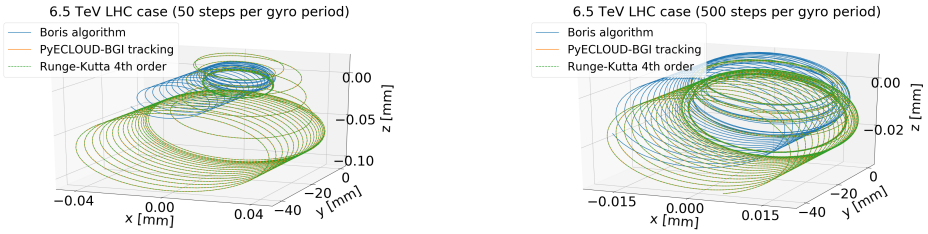

Figure 21: Trajectories emerging from the different tracking algorithms for the LHC case. Left: Using 50 time steps per gyro period. Right: Using 500 steps per gyro period.

an underestimation of the field strength. Consequently this leads to a deceleration of the particle which is too small. Because of the external magnetic field the particle circulates in a repeated fashion around $x = 0$ and is periodically subjected to this over- and underestimation of acceleration and deceleration, respectively. This leads to a significant increase in gyro momentum and therefore in gyroradius which is only due to the error in field estimation. The same holds for the E $\times$ B-drift. While the particle would actually drift in one direction on one side of the bunch center it would drift in the opposite direction on the other side of the bunch center because of the inverse field value. This opposite movement should compensate the initial drift (by drifting the same distance backwards) and this can be observed for the Boris algorithm and for the larger number of time steps (see figure 21). However if the field is over- and underestimated this leads to the situation that the drift on one side of the bunch center is larger than on the other side and thus leads to a net E $\times$ B-drift which is nonzero.

One can ask why the Boris algorithm performs better already for larger time step sizes. As explained in section 4.2.2 the Boris algorithm has been implemented with a shift of $-\Delta t/2$ of the momentum with respect to the position. That is for each velocity update the algorithm uses the field from "in between" the current value and the updated value and because the presented field shows a strong linear behavior close to $x = 0$ taking the average field is a very good approximation (see figure 22). So for the velocity update instead of overestimating (underestimating) the field the Boris algorithm uses an average value and because the updated velocity is linear in the electric field this introduces no error at all.

The quality of the uniformity assumption of the beam fields during a push also depends on the gyro momentum because this is deciding for the distance that the particle travels during the push. For that reason it is useful to observe the trajectories of a few dedicated particles before deciding for a certain time step size. By varying the time step size one should observe the trajectories converge and thus knows that a sufficiently small time step size has been reached.

PS case This behavior occurs for particles that are confined in an external magnetic field and which therefore periodically move in the linear region of the beam electric field. For the PS benchmark case no magnetic field is used and so the trajectories and the type of distortion will be significantly different than for the LHC case.

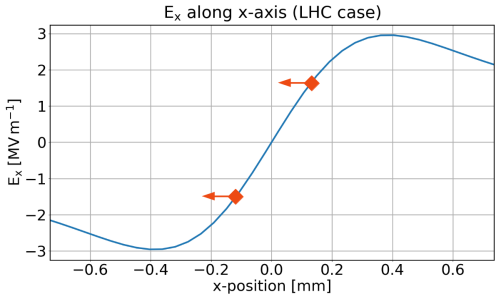

Figure 22: The electric field of a bunch. For particles moving towards the bunch (right) the electric field is overestimated and for particles moving away from the bunch (left) the field is underestimated.

Figure 23 shows the trajectory for the 3 kV PS case for a similar test particle as for the LHC case. One can observe the particle being attracted by the electric field of the beam and thus moving towards $x < 0$. In the beginning the beam field even exceeds the external electric field and hence the particle is moving "upwards", opposite to the electric guiding field. The time step size for those plots corresponds to the 50 steps per gyro period as for the LHC case, i.e. $\Delta t \approx 3.57\,\text{ps}$. This induces a displacement of $\Delta x \approx 22\,\mu\text{m}$ for the PyECLOUD-BGI tracking and the Runge-Kutta algorithm with respect to the Boris algorithm which is due to the electric field over- and underestimation, i.e. an algorithmic error. Choosing the time step size small enough is equally important here and one can use trajectories of dedicated particles as an indicator for a sufficiently small time delta.

To see the effect on the tracking accuracy with magnetic field the same case has been run for $B_y = 0.2\,\text{T}$ which is the magnetic field strength that was ultimately used for the PS IPMs. Figure 24 shows the resulting trajectories for the above test particle with $\Delta t \approx 3.57\,\text{ps}$. At the detector level ($y = -35\,\text{mm}$) the different tracking models show a deviation of $\Delta x \approx 15\,\mu\text{m}$ which is due to the quality of the field estimation. The right plot shows the deviation in dependency on the simulation time step, i.e. its evolution during the simulation. One can observe that the deviation reflects the oscillating movement of the particle and shows an increasing trend. This is due to the fact that the difference in phase advance between the algorithms grows during the simulation; if the particle was tracked further one could observe the deviation reaching a maximum value equaling the diameter of the movement and then decreasing again as the phase advance increases further.

For both cases, with and without magnetic field, the deviations diminished for $\Delta t \approx 0.357\,\text{ps}$.

5.2. Bunch electric field models

5.2.1. LHC case

Figure 25 shows the bunch electric field for the LHC case as computed by the different bunch electric field models. The *SymmetricGaussian* and the *ParabolicEllipsoid* models cannot directly be used for the presented case because both require rotational symmetry

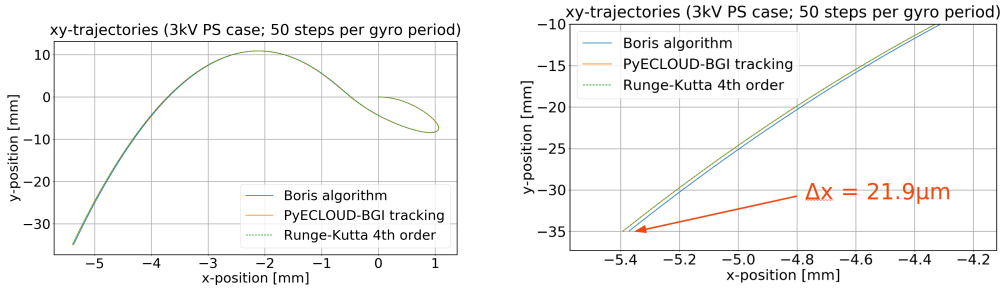

Figure 23: Trajectories emerging from the different tracking algorithms for the 3 kV PS case. Left: complete trajectory. Right: zoom of the trajectory's part showing the deviation.

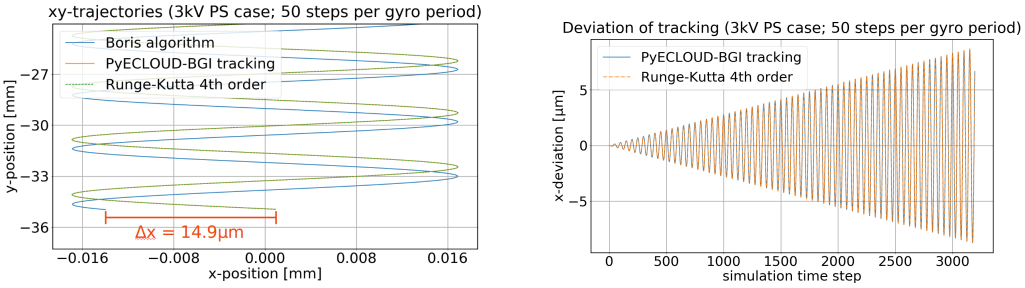

Figure 24: Left: Trajectories emerging from the different tracking algorithms for the 3 kV PS case with a magnetic field strength of 200 mT. Right: Deviation of x-position for the PyECLOUD-BGI tracking and the Runge-Kutta algorithm with respect to the Boris algorithm. The deviations reflect the oscillating movement of the particle. The final value of the deviation plot on the right is different from the one indicated in the plot on the left because the tracking algorithms also involve an deviation in y-position. That is the simulation time step when the particle reached the detector level at $y = -35$ mm is different for each algorithm.

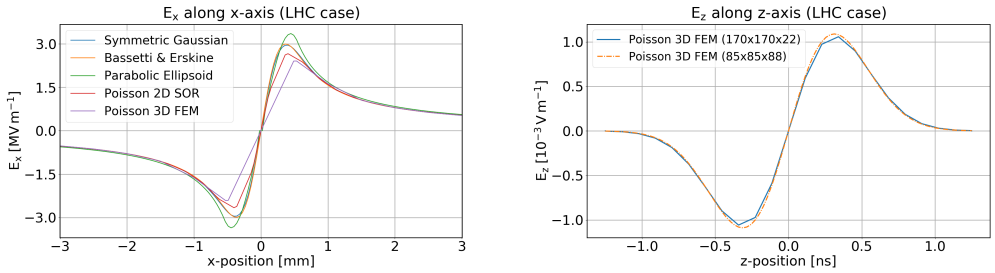

Figure 25: Bunch electric fields emerging from the different field models for the LHC case. Left: Electric field in x-direction along x-axis. Right: Longitudinal field in z-direction along z-axis.

around the z-axis (that is the width in x-direction and the height in y-direction must be similar). However, for the purpose of testing, their results have been included and are expected to deviate because of the necessary additional assumptions. The presented case involves $\sigma_x = 229\,\mu\text{m}$ and $\sigma_y = 257\,\mu\text{m}$ and therefore an average transverse size will be used for those models. In addition the bunch shape for the Parabolic Ellipsoid model must be an ellipsoid however the presented case involves a Gaussian charge distribution. Therefore we apply the following approximations. For the Symmetric Gaussian model an average width of $\sigma = (\sigma_x + \sigma_y)/2$ is used. For the Parabolic Ellipsoid model an ellipsoid with the semi-major axis $a = \sqrt{5}\sigma_z$ and the semi-minor axis $b = \sqrt{5}(\sigma_x + \sigma_y)/2$ is used. The scaling factor $\sqrt{5}$ has been chosen arbitrarily in order to match the parabolic with the Gaussian distribution. Figure 26 shows the resulting charge density for the Gaussian and the Parabolic Ellipsoid charge distributions. The electric field shows good agreement between the different models considering the approximations that were made. For the longitudinal field the three-dimensional Poisson FEM solver has been used with different grid sizes in order to estimate the influence of the (longitudinal) grid spacing. The electric field was found to remain similar also for larger grid spacings with respect to the longitudinal axis. Although the Parabolic Ellipsoid model provides three-dimensional electric fields as well, previous investigations have found that the model yields erroneous values for very long bunches [29] (note that $\sigma_z/\sigma_x \approx 2.67 \times 10^6$ in the rest frame of the bunch). The field values obtained for the LHC parameters are found to be too large and also show heavy oscillations as can be seen from figure 27. The same figure shows that using 128-bit floating point precision (the numpy library [20] offers a corresponding data type) clears the oscillations however the values remain unphysical. Thanks to the mpmath library [30] the field could be computed with even higher precision (336 bit, 100 decimal places) and by doing so it has been found that the values agree with the result that has been obtained with the Poisson FEM solver (considering the different underlying charge distributions; for the Parabolic Ellipsoid model the charges are located closer to the z-axis especially at the head and the tail of the bunch hence the larger field). This shows that the problem is solely of numerical (rather than algorithmic) nature and a solution is being investigated.

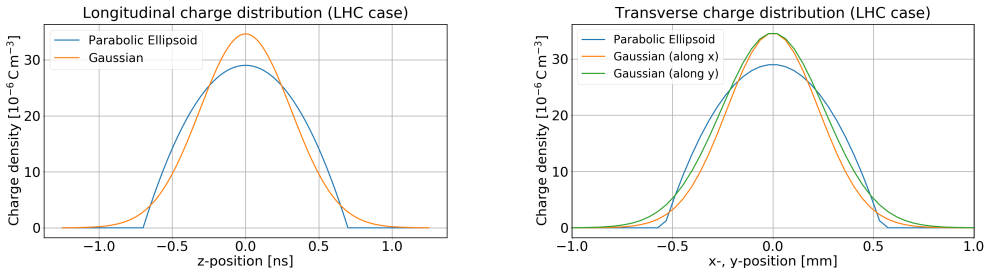

Figure 26: Charge densities for the LHC case, approximated by a Gaussian and a Parabolic Ellipsoid charge distribution. Left: Longitudinal density along z-axis. Right: Transverse density along x- and y-axis.

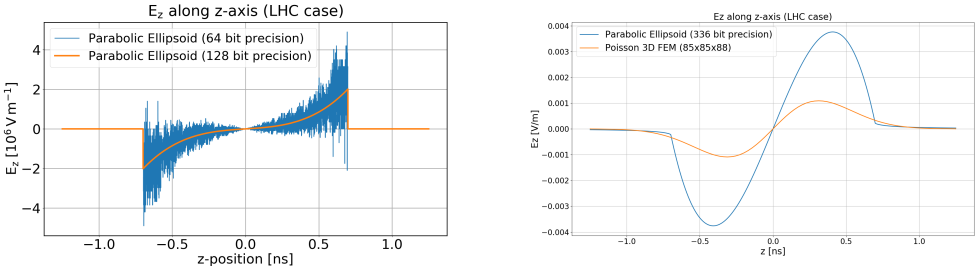

Figure 27: The longitudinal electric field computed by the Parabolic Ellipsoid model. Left: Using 64-bit and 128-bit floating point precision. Right: Using 336-bit floating point precision (100 decimal places) for the Parabolic Ellipsoid model.

5.2.2. PS case

Figure 28 shows the bunch electric field for the different models for the PS case. The same approximations as for the LHC case have been made for the Symmetric Gaussian model and the Parabolic Ellipsoid model. Because the beam is highly asymmetric in this case ($\sigma_x/\sigma_y \approx 2.64$) the quality of the assumptions is worse (see figure 29). This can be seen from the larger deviation of the field values for those models compared with the LHC case. The longitudinal electric field is larger for the Parabolic Ellipsoid model because the charges are located closer to the z-axis especially at the head and the tail of the bunch (while for the Gaussian distribution the transverse distribution is independent of the longitudinal position along the bunch).

5.3. Profile comparison

The PyECLOUD-BGI simulation code [7] was specifically written for studies of the LHC case while the JPARC-code was dedicated to the PS cases [8]. The PyECLOUD-BGI code uses a two-dimensional formulation of the beam electric field with regard to the highly relativistic beam. The JPARC-code uses external field maps for the electric guiding field because the design of the high voltage case for the PS IPMs does not include any side electrodes. For those reasons the comparison of profile distortion is made with those two codes, respectively.

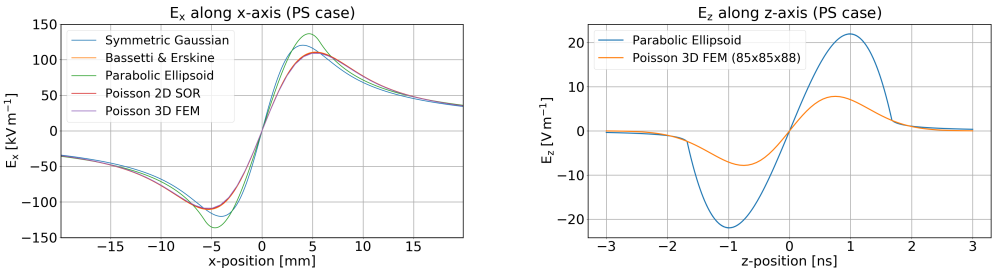

Figure 28: Bunch electric fields emerging from the different field models for the PS case. Left: Electric field in x-direction along x-axis. Right: Longitudinal field in z-direction along z-axis.

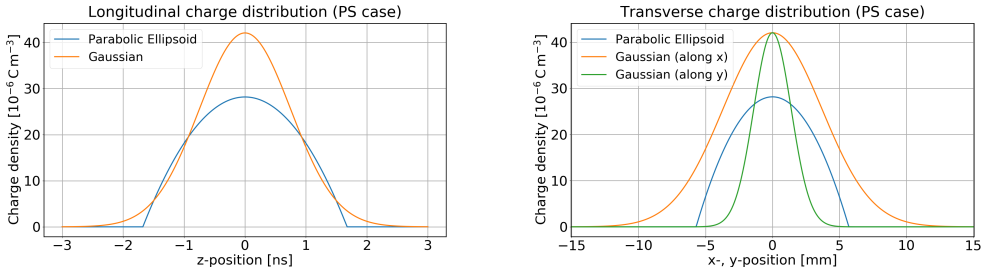

Figure 29: Charge density for the PS case, approximated by a Gaussian and a Parabolic Ellipsoid charge distribution. Left: Longitudinal density along z-axis. Right: Transverse density along x- and y-axis.

5.3.1. LHC case

Figure 30 shows the results obtained with the new simulation tool compared to the results from the PyECLOUD-BGI simulation tool as a reference. The same models as implemented by PyECLOUD-BGI have been enabled for the comparison in order to check a successful migration of these models. The right plot shows that using a larger time step size of $\Delta t = 3.57\,\mathrm{ps}$ does not affect the simulated profile in a strong way. As one can see from figure 21 the larger time step size results in an increase of gyroradius by approximately 20 µm and thus the overall effect on the measured profile is rather small. Those results confirm that the corresponding methods have been successfully migrated and implemented.

5.3.2. PS case

The results for the 3 kV and 20 kV PS cases are shown in figure 31. Again the same models as implemented by the JPARC simulation code have been enabled in order to verify a successful migration. Both results agree very good. The peculiar shapes of the simulated profiles can be explained by the fact that the measured electrons are attracted towards the center of the bunch and for smaller extraction field strengths they even move "to the other side" of the bunch causing a profile broadening (i.e. they cross the $x = 0$ point). For the 20 kV case the time of flight of the electrons is much shorter and (being attracted towards $x = 0$)

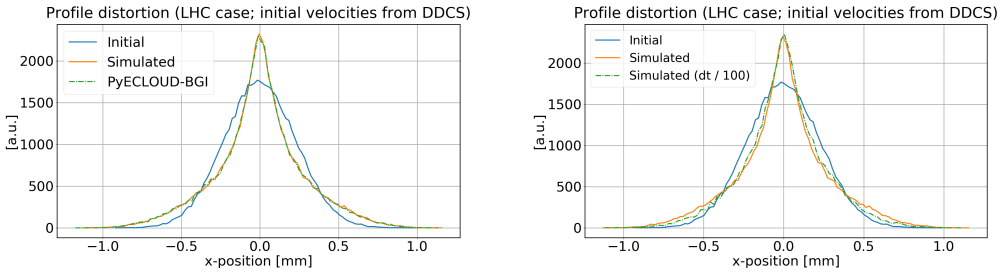

Figure 30: Profile comparison for the LHC case. For the Virtual-IPM simulation tool the same models as implemented by the PyECLOUD-BGI simulation code have been used. Left: Profile comparison for $\Delta t = 3.57\,\text{ps}$. Right: Profile comparison for $\Delta t = 0.0357\,\text{ps}$. Because for the larger time step size of 3.57 ps the error in the field estimation is bigger and the corresponding profile shows a slightly larger broadening.

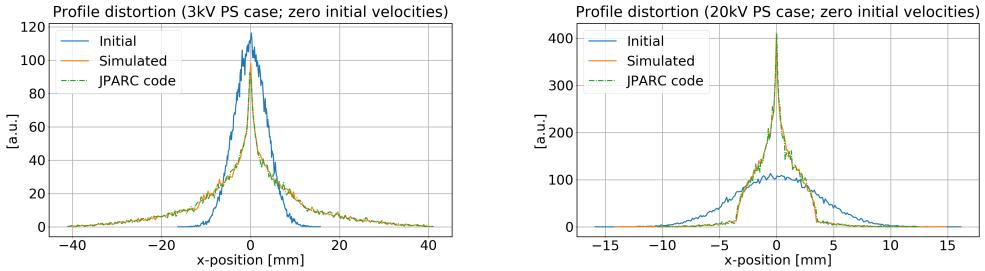

Figure 31: Profile comparison for the 3 kV and 20 kV PS cases. For the Virtual-IPM simulation tool the same models as implemented by the JPARC simulation code have been used. Left: Profile comparison for 3 kV. Right: Profile comparison for 20 kV.

they accumulate in this region. This can be seen from figure 32 which shows a mapping of initial x-positions to final x-positions. For the 3 kV case many electrons move from a positive x-position to a much larger negative one and vice versa. For the 20 kV case the accumulation near $x = 0$ can be observed.

5.3.3. SIS-18 measurements

The measurements at SIS-18 were performed using four different devices. Two devices are equipped with an electronic readout system which consists of 64 wires of 1.5 mm width. Two neighboring wires are separated by a distance of 0.6 mm resulting in a resolution of 2.1 mm. These IPMs were operated in ion detection mode and – because of the expected small distortion – served as a reference for the electron measurements. The other two IPMs are equipped with an optical readout system consisting of a multi-channel plate, a phosphor screen and a camera. The resolution of this system is 0.0755 mm per pixel (in the beam location). Those IPMs were configured to measure electrons. The beta functions at the locations of the ion and the electron IPMs are 6.95 m and 6.56 m, respectively. This difference has to be taken into account when comparing the profiles of the different devices ($\sigma \propto \sqrt{\beta}$).

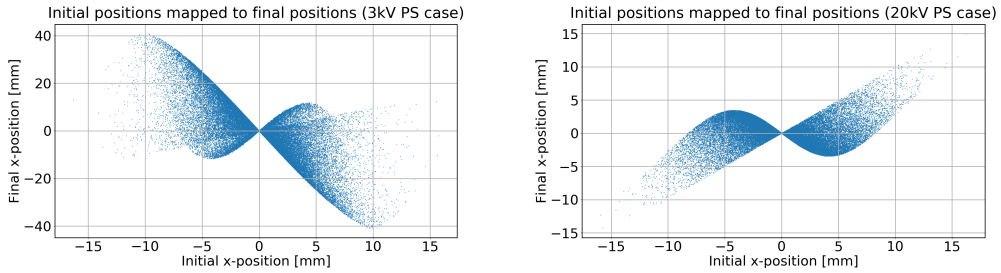

Figure 32: Mapping of initial to final x-positions for the 3 kV and 20 kV PS cases. Left: 3 kV case. Right: 20 kV case.

Because the SIS-18 served multiple users at that time it accelerated multiple beams in short cycles (the typical durations of one cycle range from a few hundred milliseconds to a few seconds). Therefore it was crucial to select only measurements from the dedicated beam. The ion IPM is equipped with a trigger system which allowed for limiting the measurements however the electron IPM was operated without trigger system. A long flat top of 6 s has been adjusted in order to be able to identify the beam by a corresponding series of similar images and to match them with the measurements from the ion IPM.

Figure 33 shows the profile image as recorded by the camera from the phosphor screen for the vertical device. A region of interest is used in order to extract the beam profile. Figure 34 shows the beam profiles that have been obtained with the different devices as well as the results from a corresponding simulation. The simulation was tuned to use the Bassetti-Erskine bunch field model (section 4.5.2) because of the large bunch length ($\sigma_z/\sigma_x \approx 531$ in the bunch frame). The validity of neglecting the longitudinal field has been confirmed by running a cross-check with the three-dimensional Poisson solver (section 4.5.5). The composition of the SIS-18 rest gas is dominated by H_2 and for that reason the Voitkiv double differential cross section (section 4.1.4) for hydrogen has been used for the initial momentum generation (the cross section is applicable for $v \lessapprox Z \ll v^2\gamma; 1 \ll v < c$, in atomic units, Z being the charge number of the projectile; both conditions are fulfilled for the presented case). Figure 35 illustrates the initial momenta of ionized particles as used for the simulation. The energy distribution shows that most particles will have energies smaller than 30 eV. The scattering angle distribution shows a slight skew favoring smaller scattering angles, that is it accounts for a "dragging" effect of the projectile on the electron that becomes pronounced due to the high charge number of the projectiles.

The simulated profile shows a slight broadening with respect to the measured electron and ion profile. Because the beam current measurements which accompanied the IPM measurements were not completely clear about the bunch population but involved some uncertainties, the bunch population might actually range from 2.0×10^7 to 2.0×10^9. Running another simulation for 2.0×10^9 ions per bunch yields results that agree much better with the measurement (see figure 34) suggesting that the actual bunch population was of this order.

Figure 36 shows the image as recorded by the camera of the horizontal device. A crack in the glass plate of the viewport is clearly visible. On the left side one can observe a small

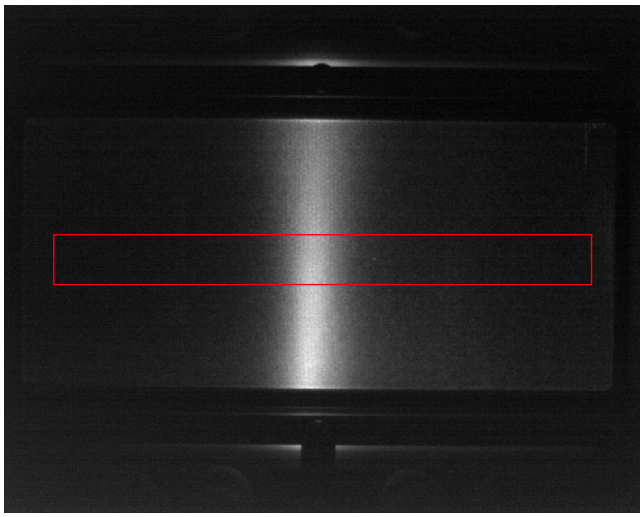

Figure 33: Measurement of the vertical beam size at SIS-18 with an extraction voltage of 8 kV. The figure shows the image as recorded by the camera. The MCP and phosphor screen assembly holders are visible at the boundaries. The region of interest from which the beam profile is extracted is marked by the red rectangle. The image shows a slight broadening at the top and bottom of the window; this effect is currently under investigation.

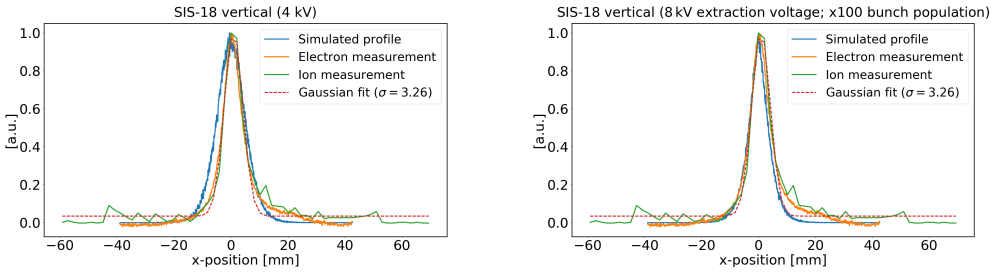

Figure 34: Measurement of the vertical beam size at SIS-18 with an extraction voltage of 8 kV. The beam profiles obtained from the ion and the electron IPMs are plotted. The measurement from the ion IPM – serving as a reference – is fitted with a Gaussian distribution ($\sigma = 3.26\,\mathrm{mm}$). The electron case has been simulated and the results are compared to the profiles. Left: Simulation for the initially assumed 2.0×10^7 ions per bunch. Right: Simulation for 2.0×10^9 ions per bunch.

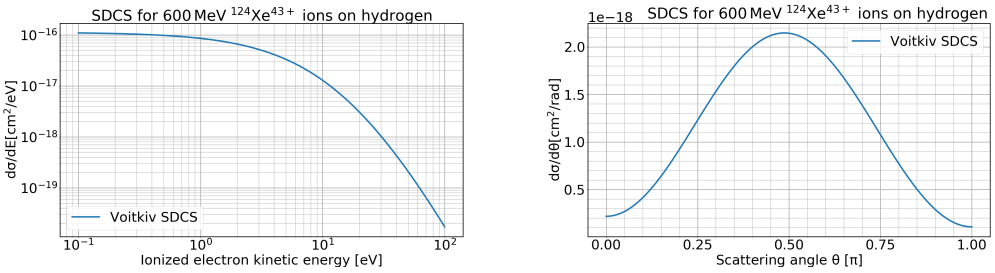

Figure 35: The initial momenta used for the SIS-18 simulations. Left: Single differential cross section with respect to energy. Right: Single differential cross section with respect to scattering angle.

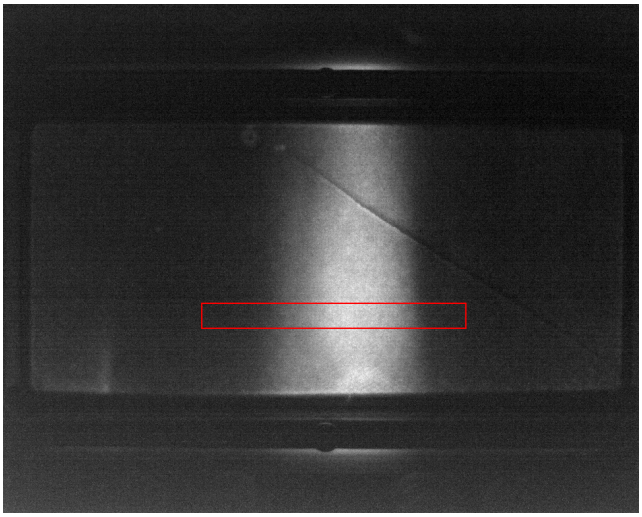

Figure 36: Measurement of the horizontal beam size at SIS-18 with an extraction voltage of 8 kV. The figure shows the image as recorded by the camera. The MCP and phosphor screen assembly holders are visible at the boundaries. The region of interest from which the beam profile is extracted is marked by the red rectangle. The image shows a crack in the center of the window as well as a stray signal on the left side. A small overlaid stray signal can be observed at the bottom of the window; these effects are currently under investigation.

stray signal and on the bottom the signal seems to be overlaid with another stray signal. The region of interest is chosen in order to exclude these effects when extracting the beam profile. Figure 37 shows the beam profiles obtained from the different devices together with the results from the simulation. The same models as for the vertical measurement have been used for the simulation and again the simulation involves a broadening of the profile. For this case the measured electron profile is actually more narrow than the reference ion profile. This narrowing suggest that again space charge effects play a role. Checking the results for another simulation for 2.0×10^9 ions per bunch shows that the profiles agree much better (see figure 37), also suggesting that the beam current has been underestimated.

In summary the comparison of simulation results with the measurement data shows a reasonable agreement, considering the uncertainty for the bunch population.

5.4. Performance

The tests have been performed on a machine with a Intel Core i7-5500U @ 2.40GHz x 4 CPU and a 2 x 8GB SO-DIMM DDR 1600MHz memory equipment. Typical scales such as 50 000 particles and 3000 time steps have been used for estimating the performance.

5.4.1. Particle tracking

Figure 38 shows the performance results for the particle tracking models. The PyECLOUD-BGI model is the fastest because it simply requires plugging into a predefined formula. The

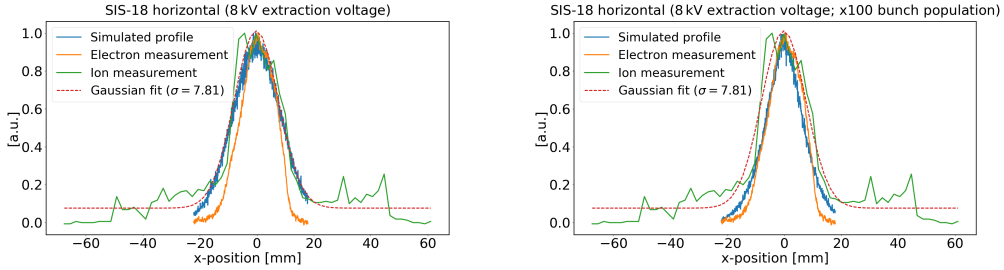

Figure 37: Measurement of the horizontal beam size at SIS-18 with an extraction voltage of 8 kV. The beam profiles obtained from the ion and the electron IPMs are plotted. The measurement from the ion IPM – serving as a reference – is fitted with a Gaussian distribution ($\sigma = 7.81\,\mathrm{mm}$). The electron case has been simulated and the results are compared to the profiles. Left: Simulation for the initially assumed 2.0×10^7 ions per bunch. Right: Simulation for 2.0×10^9 ions per bunch.

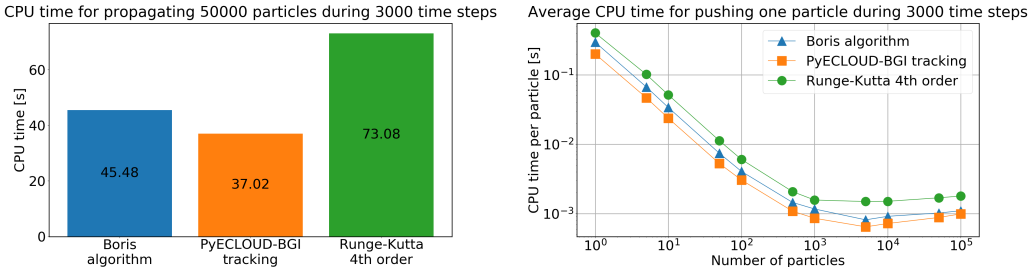

Figure 38: Performance of the different particle tracking models. Left: Total CPU time needed to push 50 000 particles during 3000 time steps. Right: Average CPU time required for pushing one particle during 3000 time steps in dependence on the number of particles issued for an update each time step.

Runge-Kutta algorithm is the slowest because of its four intermediary evaluations of the acceleration term. One can also observe that the procedure of updating particles involves some overhead whose relative contribution diminishes for a large number of particles (i.e. the actual time required for updating dominates).

5.4.2. Bunch electric field models

Figure 39 shows the performance results for the bunch electric field models. The Poisson FEM model uses significantly higher CPU times because the evaluation of the electric field is done for a single position at a time and the different positions are looped over in Python (this is due to the API of the used FEM framework accepting a single position rather than an array of positions). Because all other models use the numpy [20] and scipy [37] libraries for the computational work and because those libraries provide APIs for array processing the particles' positions and velocities are looped over within the compiled and optimized functions of those libraries which leads to a significant increase in efficiency.

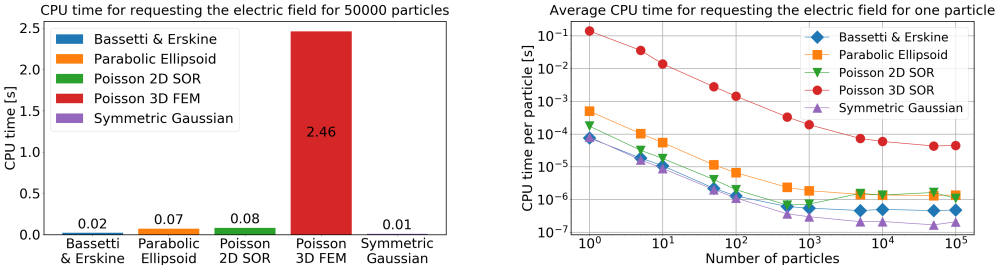

Figure 39: Performance of the different bunch electric field models. Left: Total CPU time needed to request the electric field for 50 000 particles. Right: Average CPU time required for requesting the field of one particle in dependence on the number of particles for which the field has been requested simultaneously.

SOR Poisson solver Besides the CPU time required for retrieving the fields the SOR Poisson solver also requires time for (numerically) solving Poisson's equation. For the PS case a 280×280 grid was used (resulting in a grid spacing of 0.25 mm). The over-relaxation parameter was set to 1.9 and a convergence limit of 1.0×10^{-6} was used. With those settings the model needed 2818 iterations and 13 minutes to find a solution. For the LHC case using the same values for the over-relaxation parameter and the convergence limit together with a 340×340 grid (0.25 mm grid spacing) the model finished computation after 3669 iterations and 22 minutes.

FEM Poisson solver The FEM Poisson solver also requires CPU time for solving Poisson's equation. Using Lagrange polynomials of degree 1 and a $170 \times 170 \times 22$ grid (resulting in a transverse grid spacing of 0.41 mm and a longitudinal grid spacing of 0.27 ns) the model needed 5 minutes and 6 GB of memory in order to find a solution for the PS case. For the LHC case also a $170 \times 170 \times 22$ grid was used (resulting in 0.5 mm transverse and 0.11 ns longitudinal grid spacing) and the model finished computation after 11 minutes using 7 GB of memory.

6. Summary and Conclusions

A new simulation framework, "Virtual-IPM", has been developed in order to structurize and unify various aspects needed for Ionization Profile Monitor simulations. The presented code contains a complete set of algorithms, making it fully functional. Because of the modular structure of the code the development of additional modules is strongly facilitated which allows the application to be extended to other use cases. Future intended use cases include simulations of Beam Induced Fluorescence monitors as well as simulations of Electron Wire Scanners and investigations on the influence of supersonic gas jets.

The main components of the framework are *Particle Generation*, *Particle Tracking*, *Bunch Shapes* and *Bunch Electric Fields*, *Guiding Fields* and *Particle Detection*. The methods which are currently implemented include:

- Particle Generation:
  - Ionization described by single and double differential ionization cross sections
- Particle Tracking:
  - Boris algorithm
  - Runge-Kutta 4th order
  - Analytical solution for specific field configurations
- Bunch Shapes:
  - Gaussian charge distribution
  - Parabolic ellipsoidal charge distribution
- Bunch Electric Fields:
  - Analytical, two-dimensional solution for a symmetric Gaussian charge distribution
  - Analytical, two-dimensional solution for an asymmetric Gaussian charge distribution
  - Analytical, three-dimensional solution for a rotational-symmetric parabolic-ellipsoidal charge distribution
  - Numerical, two-dimensional Poisson solver based on the Successive Over-Relaxation method for arbitrary charge distributions
  - Numerical, three-dimensional Poisson solver based on the Finite Elements method for arbitrary charge distributions
- Guiding Fields:
  - Uniform fields
  - Two- and three-dimensional field maps

- Particle Detection:
  - Ionization Profile Monitor model

The application can simulate single-bunch as well as multi-bunch scenarios, the latter becoming important for the tracking of ions.

The different components and implementations of the framework have been tested and benchmarked against existing results in order to verify their integrity. All models have been found to work correctly and estimations for their applicability to different usage scenarios have been given. Because of the limited availability of adequate experimental data the code was only partially benchmarked against measurements. Future benchmarking with measurement data is foreseen.

The application includes a graphical user interface which can be used to configure, run and post-process complete simulations. During configuration the user may choose which particular solutions they want to apply to the different part problems of a simulation such as particle generation or beam field computation. The application can be controlled equally from the command line as well as from the graphical user interface. It is accompanied by various command line tools that help convert input and output data to and from the application in order to be compatible with existing standards.

The application and the code are adequately documented which helps both users and developers of the application to get involved. It is available as a Python package on the Python package index, see [38]. The code is publicly available as a git repository and is open for collaboration, see [39]. The git repository also includes an issue tracker for reporting bugs or discussing new features or enhancements. The documentation is available as an HTML website, see [40].

In the near future the application will be presented to the beam instrumentation community at the International Beam Instrumentation Conference (IBIC). In addition a number of developments and upgrades of the code are in work. The application is being used for various studies.

References

[1] D. Bodart, B. Dehning, S. Levasseur, P. Pacholek, A. Rakai, M. Sapinski, K. Satou, G. Schneider, D. Steyart, and J. W. Storey, "Development of an Ionization Profile Monitor based on a pixel detector for the CERN Proton Synchrotron," *Proceedings of IBIC2015, TUPB059*, 2015.

[2] "Beam Gas Ionization Monitors." http://bgi.web.cern.ch/.

[3] D. Vilsmeier, "Profile distortion by beam space-charge in Ionization Profile Monitors," *CERN-THESIS-2015-035*, 2015.

[4] D. Vilsmeier, B. Dehning, and M. Sapinski, "Investigation of the effect of beam space-charge on electron trajectories in Ionization Profile Monitors," *Proceedings of HB2014, MOPAB42*, 2014.

[5] M. Sapinski *et al.*, "Ionization Profile Monitor simulations – status and future plans," *Proceedings of IBIC2016, TUPG71*, 2016.

[6] C. A. Thomas, F. Belloni, and J. Marroncle, "Space charge studies for the Ionization Profile Monitors for the ESS cold LINAC," *Proceedings of IBIC2016, TUPG81*, 2016.

[7] G. Iadarola, "Electron cloud studies for CERN particle accelerators and simulation code development," *CERN-THESIS-2014-047*, pp. 34–38, 2014.

[8] K. Satou, "Python based particle tracking code for monitor design." https://indico.cern.ch/event/491615/contributions/1169450/attachments/1237859/1818556/presentation_file_final.pdf. IPM Workshop 2016.

[9] R. Thurman-Keup, "IPM simulations at Fermilab." https://indico.cern.ch/event/491615/contributions/1169451/attachments/1237853/1818545/IPM_Simulation_Workshop.pdf. IPM Workshop 2016.

[10] R. Williamson *et al.*, "IPM simulation and correction with strong space charge." https://indico.cern.ch/event/491615/contributions/1169454/attachments/1238153/1819099/IPM_RWilliamson_2016_final.pptx. IPM Workshop 2016.

[11] S. Udrea and P. Forck, "Bunch: A tracking code based upon analytic field computations in use at GSI." http://indico.gsi.de/getFile.py/access?contribId=12&sessionId=11&resId=0&materialId=slides&confId=5366. IPM Workshop 2017.

[12] C. C. Wilcox, B. Jones, A. Pertica, and R. E. Williamson, "An investigation into the behaviour of residual gas Ionisation Profile Monitors in the ISIS extracted beamline," *Proceedings of IBIC2016, WEPG68*, 2016.

[13] F. Becker, "Non-destructive profile measurement of intensive heavy ion beams," *urn:nbn:de:tuda-tuprints-23328, http://tuprints.ulb.tu-darmstadt.de/2332*, 2010.

[14] H. D. Zhang *et al.*, "A supersonic gas-jet based Beam Induced Fluorescence prototype

monitor for transverse profile determination," *Proceedings of IPAC2017, MOPAB139*, 2017.

[15] K. Satou, "News from J-PARC: the system, the data and the simulation code." http://indico.gsi.de/getFile.py/access?contribId=11&sessionId=1&resId=1&materialId=slides&confId=5366. IPM Workshop 2017.

[16] W. Blokland and S. Cousineau, "A non-destructive profile monitor for high intensity beams," *Proceedings of 2011 Particle Accelerator Conference, WEOCN2*, 2011.

[17] G. Stancari *et al.*, "Conceptual design of hollow electron lenses for beam halo control in the Large Hadron Collider," *CERN-ACC-2014-0248, FERMILAB-TM-2572-APC*, 2014.

[18] G. Iadarola, H. Bartosik, M. Driss-Mensi, H. Neupert, G. Rumolo, and M. Taborelli, "Recent electron cloud studies in the SPS," *Proceedings of IPAC2013, WEPEA014*, 2013.

[19] "Ionization cross sections package." https://pypi.python.org/pypi/ionics.

[20] "NumPy." http://www.numpy.org/.

[21] "Anna." https://pypi.python.org/pypi/anna. version 0.3.

[22] "ReactiveX for Python." https://pypi.python.org/pypi/Rx.

[23] A. B. Voitkiv, N. Gruen, and W. Scheid, "Hydrogen and helium ionization by relativistic projectiles in collisions with small momentum transfer," *J.Phys.B: At.Mol.Opt.Phys*, vol. 32, 1999.

[24] J. P. Boris, "Relativistic plasma simulation-optimization of a hybrid code," *Proceedings of the Fourth Conference on Numerical Simulation of Plasmas*, pp. 3–67, 1970.

[25] "CST Studio." https://www.cst.com/.

[26] "Poisson Superfish." http://laacg.lanl.gov/laacg/services/download_sf.phtml.

[27] M. Bassetti and G. A. Erskine, "Closed expression for the electrical field of a two-dimensional Gaussian charge," *CERN-ISR-TH/80-06*, 1980.

[28] A. Stegun, "Handbook of mathematical functions," *Applied Mathematics Series*, vol. 55, p. 297, 1972.

[29] P. Strehl, *Beam Instrumentation and Diagnostics*. Springer, 2016. Chapter 8: The Electromagnetic Fields of Bunches.

[30] "An arbitrary floating point precision library." http://mpmath.org/.

[31] J. R. Nagel, "Solving the generalized Poisson equation using the Finite-Difference Method (FDM)," February 15, 2012.

[32] "FEniCS." https://fenicsproject.org/.

[33] H. P. Langtangen and A. Logg, "Solving PDEs in minutes - the FEniCS tutorial volume i," April 5, 2017.

[34] "Specification of the format to be used for exchange of data related to transverse beam profile measurements." https://twiki.cern.ch/twiki/pub/IPMSim/Internal_Documents/TransverseBeamProfileMonitors_DataExchangeFormat.pdf. version 0.3.

[35] "Data Analysis GUI." https://gitlab.com/IPMsim/DataAnalysis.

[36] H. Qin *et al.*, "Why is Boris algorithm so good?," *Physics of Plasmas*, vol. 20, 2013.

[37] "Scipy." https://www.scipy.org/.

[38] "Virtual-IPM on the Python Package Index." https://pypi.python.org/pypi/virtual-ipm. version 1.1.

[39] "Virtual-IPM on GitLab." https://gitlab.com/IPMsim/Virtual-IPM. version 1.1.

[40] "Virtual-IPM Documentation." http://ipmsim.gitlab.io/Virtual-IPM/. version 1.1.

Appendices

A. How to install the application

A.1. Installation

Note: Although the application is compatible with both Python 2.7 and Python 3.5/3.6 as well as PyQt4 and PyQt5 it is recommended to use it with Python 3 and PyQt5 because the support for Python 2.7 and PyQt4 eventually will be dropped.

A.1.1. Via Anaconda (recommended)

Installing Anaconda The recommended way is to install the scientific Python distribution Anaconda. It contains most of the dependencies already by default and will handle the automatic installation of all others. Make sure to select "Add Anaconda to your path" upon installation. After Anaconda has been installed successfully start the "Anaconda Navigator" (on Unix run `anaconda-navigator`; if you didn't add Anaconda to your path then you need to run `/usr/bin/env PATH=~/anaconda3/bin/:$PATH ~/anaconda3/bin/anaconda-navigator`). In the left panel navigate to "Environments".

Creating a separate environment (optional) If you're already using Anaconda or planning to use it more often you probably don't want to mix up installations of your packages and therefore create a separate environment. To do so select "Create" at the bottom of the environments panel. You can give the new environment any name but why not choose "Virtual-IPM".

The application uses numpy and scipy which are best installed from Anaconda's package index (pypi doesn't provide binaries for Windows). To do so select the environment which you just created. On the right side of the screen Anaconda shows all packages that are currently installed in this environment. Change the dropdown entry "Installed" at the top of the list to "Not installed". Search for "numpy" and "scipy" using the search field right next to it, select them by checking the corresponding boxes and install them by clicking "Apply" in the bottom right corner of the screen. In order to use the GUI you also need to install PyQt and matplotlib. Search for "pyqt" and "matplotlib" and install them similarly to numpy/scipy.

Installing the Virtual-IPM application To install the Virtual-IPM application select one of the available environments (it will be installed in this environment). Then press the ">" icon and select "Open Terminal". In the emerging terminal just type `pip install`

`virtual-ipm`. That's it! For information on how to use the application please consider *the usage instructions*. See also *Creating a desktop entry* and *Verifying the installation*.

A.1.2. Manual installation (advanced)

Virtual-IPM ships as a Python package named virtual-ipm and thus its installation is rather straightforward. Upon installation it will pull in quite a number of other Python packages as dependencies and therefore you might want to set up a separate virtualenv for usage with this application.

Requirements

General The application runs on Python which is shipped with most Unix systems. On Windows please download the latest version of Python from here. Both Python 2.7 and Python 3.x (x >= 5) are supported however using Python 3 is strongly recommended (because support for Python 2.7 eventually will be dropped).

Note: Windows users

On Windows make sure that you select the last item in the list of components that will be installed: "Add Python.exe to PATH". This option unselected by default however we'll use Python from the command line later on and for that purpose the executables must be available on the PATH.

Note: Windows users

The application depends upon a few other packages. On Unix no further steps need to be taken. On Windows however the `numpy` and `scipy` dependencies can't be installed automatically as they require precompiled binaries for the specific operating system. Scipy doesn't support binaries for Windows officially. Please consider the scipy installation instructions about scientific Python distributions and inofficial binaries.

For usage of the GUI The GUI uses PyQt which must be installed separately. In order to use the application it is not required to install the GUI (you can also run it from the command line) however it is recommended especially because the configuration process will be much easier.

If you're using Python 3.5 or greater you can install PyQt5 from the Python package index via `pip install pyqt5`. Otherwise you can downloaded the latest release from the PyQt download site. Please follow the installation instructions on this website. Usually you have to build it from source which is fairly simple though.

Note: Windows users

Alternatively to building PyQt from source you can download the inofficial PyQt binaries.

1. Download the correct PyQt version from the above mentioned website and make sure that you match your Python and OS version.
2. Open Powershell (press the Windows key and type "Powershell" in the search field; using the x86 version is recommended) and navigate to the folder where you downloaded the `.whl` file to (usually `Downloads`). Then install the package via `pip install <version>.whl` where you replace `<version>.whl` with the name of the file you downloaded (you can use tab completion in Powershell).

Installing the package You can obtain the application either via pip from the Python Package Index, by cloning the repository using `git` or by downloading the latest release as a snapshot (using your web browser for example).

Installation via pip (recommended) To install the latest release via pip open a terminal (Powershell on Windows) and run `pip install virtual-ipm` (if you want to install it into a virtualenv don't forget to activate it before running pip). That's it!

Note: Windows users

If `pip` is not found on Windows then you probably have to add your Python installation to the PATH environment variable.

Cloning the repository

1. Please consider these instructions on how to install git for your operating system.
2. Clone the repository: `git clone https://gitlab.com/IPMsim/Virtual-IPM.git`

Then you can install the application by following those steps:

1. Navigate to the application's directory: `cd Virtual-IPM`
2. Install the requirements: `pip install -r requirements.txt`
3. Install the package: `pip install -e .`

Downloading the latest release

1. Download the application as a
   - zip archive

- tar ball

2. Unpack the zip file / tar ball: `unzip Virtual-IPM-master-<sha1>.zip` or `tar xzf Virtual-IPM-master-<sha1>.tar.gz` where <sha1> is a hexadecimal number which represents the hash of the latest commit on the release branch (master).
3. Rename the unpacked folder to "Virtual-IPM": `mv Virtual-IPM-master-<sha1> Virtual-IPM`
4. Follow the installation instructions from *Cloning the repository*.

Note: Windows users

If you are on Windows an error related to path lengths may occur upon unpacking. This is because Windows has a maximum path length of 260 characters (see this document). However you can skip that error as the file in question is only part of the test suite and not used by the application core.

A.1.3. Verifying the installation

To test if the installation was successful you can run:

```
python -c "from __future__ import print_function; import virtual_ipm; print(virtual_
↪ipm.__version__)"
```

It should print the application's version number to your console.

A.1.4. Creating a desktop entry

If you are on Linux or Windows you can also create a desktop entry by running `virtual-ipm-settle`.

A.2. Updating the package

A.2.1. For installations via Anaconda

Run a terminal in the environment in which you installed the application from the Anaconda Navigator. Then just type `pip install virtual-ipm --upgrade`.

A.2.2. For manual installations

pip: If you installed the package from pypi via pip then run `pip install virtual-ipm --upgrade`.

repo: If you have obtained the application by cloning the repository you can upgrade it by simply navigating to the repository and running `git pull origin master`. That's it!

archive: If you downloaded the latest release as a snapshot then you need to uninstall the current version via `pip uninstall virtual-ipm` and then perform the same steps again using a new snapshot.

Check the new version via:

```
python -c "from __future__ import print_function; import virtual_ipm; print(virtual_
↪ipm.__version__)"
```

A.3. Uninstalling the package

No matter how you obtained the package you can uninstall it by running `pip uninstall virtual-ipm` (in Anaconda run a terminal in the corresponding environment in order to execute the command).

B. How to use the application

B.1. Conventions

`x, y, z` are used to denote the coordinate system. The beam(s) move(s) along the z-axis (in positive direction). IPM profiles are measured along the x-axis.

B.2. Via the command line

The application ships with a command line script which can be used to run simulations: `virtual-ipm`. You need to provide it a configuration file which contains all the necessary parameters for the simulation:

```
virtual-ipm /path/to/config.xml
```

For generating such a configuration file it is recommended to use the graphical user interface (at least for the first version, minor changes can be applied directly to the file of course).

Running the script will log some information to the console (mostly the simulation progress). The logging level can be controlled via the `--console-log-level` switch:

```
virtual-ipm /path/to/config.xml --console-log-level warning
```

The common Python logging levels are available. The console output can be suppressed completely with the `--quiet-console` switch. For more options and additional parameters consider the script itself:

```
virtual-ipm --help
```

B.3. Via the GUI

The GUI can be started by running `virtual-ipm-gui`.

B.3.1. Configuration

The GUI can be used to generate configuration files. You need to fill in all the input forms and you can save a setting via the menubar item `File -> Save as -> XML`. This will save the current configuration as an XML file. Figure 40 shows a screenshot of the configuration part of the GUI.

Some parameter types support special syntax for entering values:

- Integers: You can sum (`+`), subtract (`-`), multiply (`*`) or raise integers to a power (`**`); the common calculation rules apply. So `4 + 2 * 2**2 - 2` is a valid specification and evaluates to `10`.
- Numbers: Numbers support all common calculation rules as well as numpy functions/constants via the prefix `numpy.` or `np.`. So `2 * numpy.pi + np.cos(np.pi/2)` is a valid specification.
- Physical quantities: You need to specify a magnitude which supports all features of a number as well as allows you to access members of `scipy.constants` and values in `scipy.constants.physical_constants`. To do so you can use the following syntax `%(<name>)` where `<name>` is either the name of a member of `scipy.constants` or a key of `scipy.constants.physical_constants`. So `%(electron mass energy equivalent in MeV) * 1.0e6 / %(speed_of_light)**2 * %(elementary charge)` is a valid specification. In addition you need to select a unit from the dropdown menu on the right.

If you want to modify an XML configuration file directly please note the following structure of the XML elements:

- `<NameOfParameter unit="...">...</NameOfParameter>`
- A unit must be given for physical quantities (and vectors or tuples thereof); common abbreviations such as `"m"` for meters or `"V/m"` for volt per meter are used. If an invalid unit was given this will be reported when the simulation is started.
- Note that the elements of vectors or tuples need to be enclosed in square brackets, e.g. `[1, 2, 3]`; elements can be separated by either a comma or at least two whitespaces (`[1 2 3]` is the same as `[1, 2, 3]`).

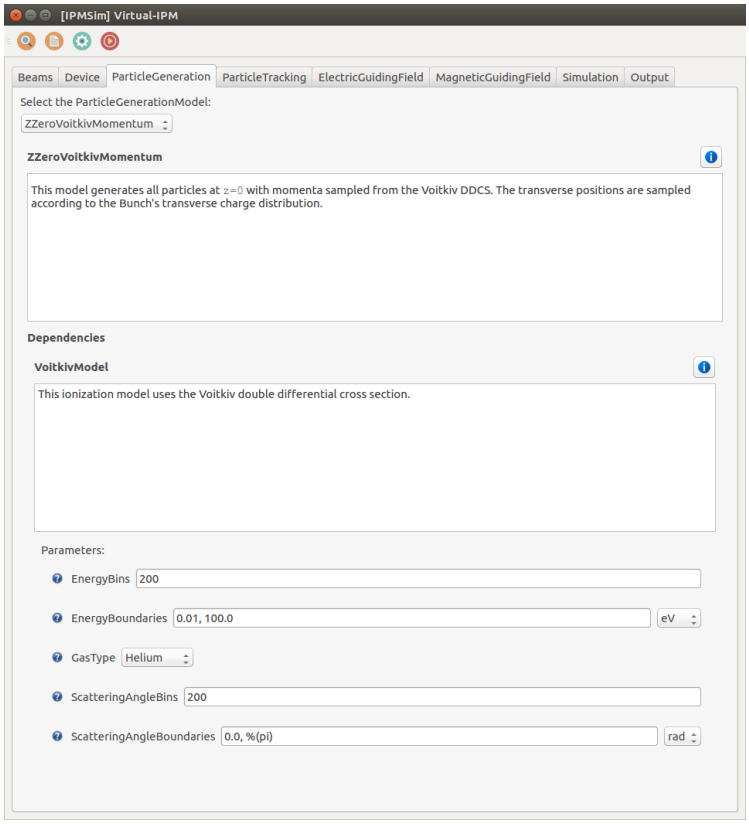

Figure 40: Screenshot of the configuration GUI.

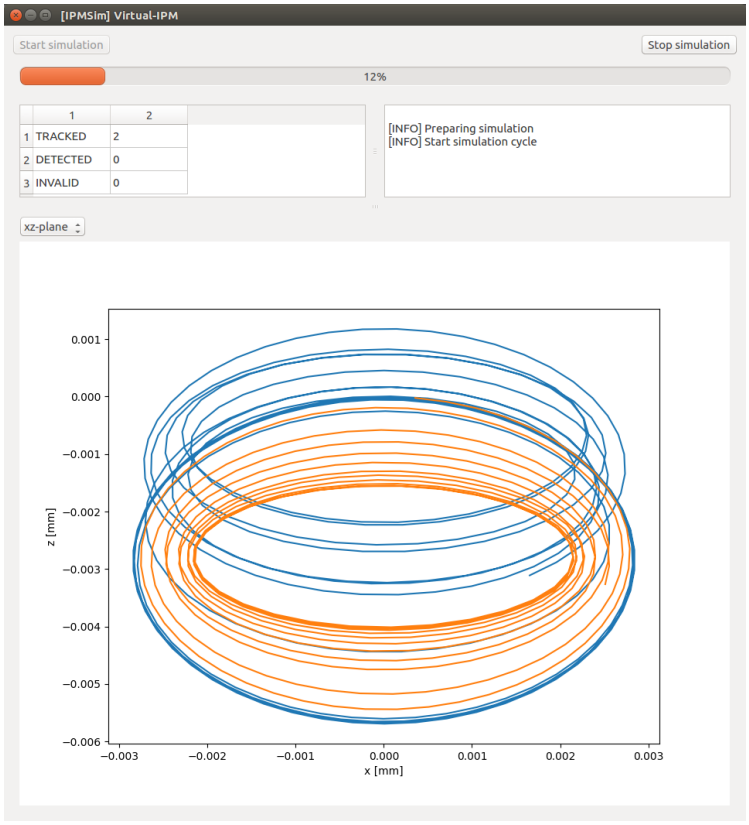

Figure 41: Screenshot of the simulation GUI.

B.3.2. Simulation

The GUI has a menubar item *Simulation* from which you can select different options. Either you can run the current configuration (this attempts to save the configuration using a suitable filename and reports if the chosen file already exists). Or you can choose to manually select a configuration file which will be used to determine the parameters of the simulation. A new window opens from which you can start the simulation by pressing the "Start simulation" button in the left upper corner. You can also add a live view for monitoring particle trajectories by pressing the button "Monitor particles" at the bottom of the window. You are then asked to enter a number of *particle IDs*. Those IDs determine which particles will be monitored. Whenever a particle is created during the simulation it is assigned an ID which starts at 0 and is incremented by 1 for each particle generated. Figure 41 shows a screenshot of this part of the GUI.

Warning: Using the monitor feature will significantly slow down the simulation (due to the graphics rendering), especially if you select multiple particles.

What happens when I start a simulation? A separate thread gets instantiated which is used to run the simulation. The following actions take place during a simulation run:

1. The configuration is loaded. If any configuration errors are encountered this will be displayed in a corresponding message box.
2. The simulation is prepared. This involves auxiliary tasks as well as any necessary pre-computations for the selected models. For example if you selected a Poisson solver for the bunch electric field it will be computed at this stage. A corresponding message appears in the log view in the upper right part of the window. Note that the progress bar won't advance during this stage but only as the main part of the simulation starts.
3. The main part of the simulation is started. This is the time step iteration and during each time step particles will be created, propagated and potentially detected. The corresponding view in the upper left region shows how many particles have a certain status at each moment. The progress bar on the top indicates how many time steps have already been performed (relative to the total number of time steps).
4. After the simulation has finished the result (the output) is finalized and written to file(s). A corresponding message appears in the log view.

B.3.3. How can I test the current configuration?

If you just want to test the current setup by for example observing a particle's trajectory you can navigate to "ParticleGeneration" and select the "SingleParticle" model. This model allows you to specify the initial parameters of one particle which will be generated during the simulation. Then just select "Simulation -> Run current configuration". In the emerging simulation window press press "Monitor particles" at the bottom and then enter `0` for the "PublishedParticles" parameter. Confirm your choice with the "Ok" button in the right bottom corner and then start the simulation. The dropdown menu above the plot view lets you specify which projection of the trajectory you want to view (see figure 41).

B.4. Output

The output of the simulation is controlled by the `OutputRecorder` component. Different recorders are available and they can be specified in the configuration file. For the beginning `BasicRecorder` is a good start, which will save the initial and final parameters of particles (it also provides a few switches for tuning which parameters should be saved). The name of the output file must be specified in the configuration as well.

B.4.1. How can I check the output?

If you wan't to take a quick look at the generated output in form of some plots you can do that via the GUI too. Just navigate to `Simulation -> Analyze results`. A new window opens from which you can open an output file (an output file that has been generated by

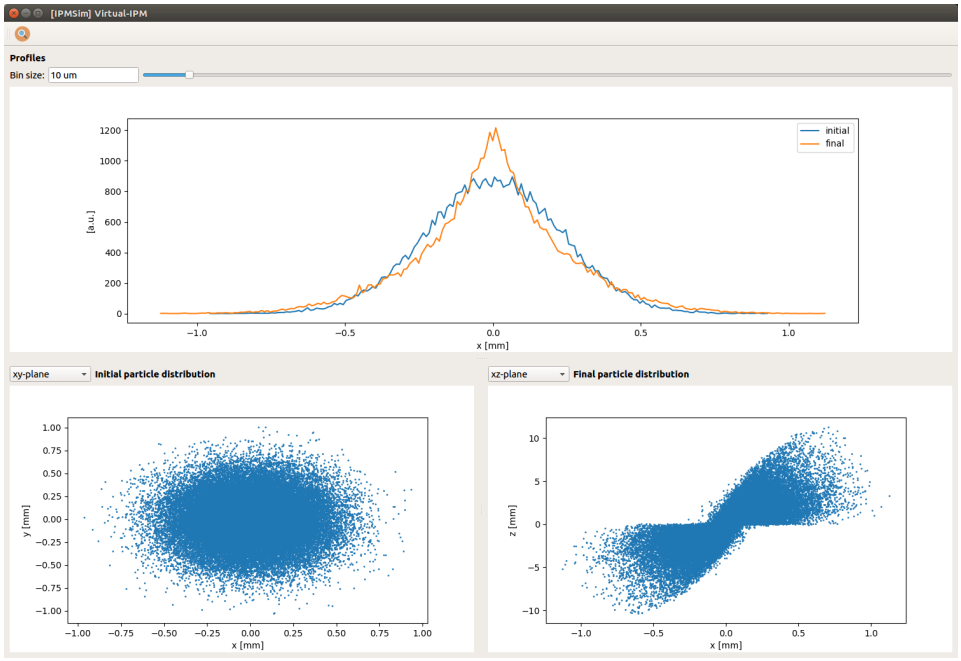

Figure 42: Screenshot of the post-analysis GUI.

`BasicRecorder`) using `File -> Open output file`. The top plot shows the initial and the final (measured / simulated) beam profiles and the two bottom plots show the initial and final (spatial) particle distributions respectively. The dropdown menu allows for switching between different distributions. Figure 42 shows a screenshot of the corresponding part of the GUI.

B.5. Command line tools

The application ships with a few command line tools which are useful for converting input and output files:

- `vipm-cst-to-csv` - This script can be used to convert a field map generated from CST studio to a CSV file in the format that is expected by the models "CSVAdaptor2D" and "CSVAdaptor3D". For more information about the format please see the corresponding models.
- `vipm-csv-to-xml` - This script converts an output file generated by `BasicRecorder` to a common XML data format which can be read and visualized by this data analysis GUI. Please note that both initial and final parameters must be present in the output file (this is the default behavior of `BasicRecorder`).