ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-FR2A004

# A PHYSICS-BASED SIMULATOR TO FACILITATE REINFORCEMENT LEARNING IN THE RHIC ACCELERATOR COMPLEX*

L. K. Nguyen†, K.A. Brown[1], M.R. Costanzo, Y. Gao, M. Harvey, J. P. Jamilkowski, J. T. Morris, V. Schoefer, Collider-Accelerator Dept., Brookhaven National Laboratory, Upton, NY, USA
[1]also at ECE Dept., Stony Brook University, Stony Brook, NY, USA

*Abstract*

The successful use of machine learning (ML) in particle accelerators has greatly expanded in recent years; however, the realities of operations often mean very limited machine availability for ML development, impeding its progress in many cases. This paper presents a framework for exploiting physics-based simulations, coupled with real machine data structure, to facilitate the investigation and implementation of reinforcement learning (RL) algorithms, using the longitudinal bunch-merge process in the Booster and Alternating Gradient Synchrotron (AGS) at Brookhaven National Laboratory (BNL) as an example. Here, an initial fake wall current monitor (WCM) signal is fed through a noisy physics-based model simulating the behavior of bunches in the accelerator under given RF parameters and external perturbations between WCM samples; the resulting output becomes the input for the RL algorithm and subsequent pass through the simulated ring, whose RF parameters have been modified by the RL algorithm. This process continues until an optimal policy for the RF bunch merge gymnastics has been learned for injecting bunches with the required intensity and emittance into the Relativistic Heavy Ion Collider (RHIC), according to the physics model. Robustness of the RL algorithm can be evaluated by introducing other drifts and noisy scenarios before the algorithm is deployed and final optimization occurs in the field.

## INTRODUCTION AND MOTIVATION

Interest in machine learning (ML) for use at particle accelerator facilities has rapidly expanded over the years, and accelerator scientists and engineers engaging in ML continue to identify and deliver on important applications [1]. However, the time and resources needed for ML development and model training can be prohibitive. At Brookhaven National Laboratory (BNL), for example, the bunch merge process in the Booster and Alternating Gradient Synchrotron (AGS) rings has been identified as a potentially high-reward area for ML optimization due to the criticality of good bunch merging to operations. Despite this, competing priorities and lack of machine availability have impeded investigations. In particular:

1. Booster & AGS are part of the accelerator chain for multiple programs, and they are often in operational use when not supplying the Relativistic Heavy Ion Collider (RHIC) with beam. Real machine time for ML development is therefore very hard to come by.
2. Any machine downtime is generally allotted to maintenance and/or needed repairs. Meanwhile, part of the ML development cycle is purely software-related (e.g., debugging). This makes real machine time for ML development expensive both in terms of financial costs and opportunity costs.

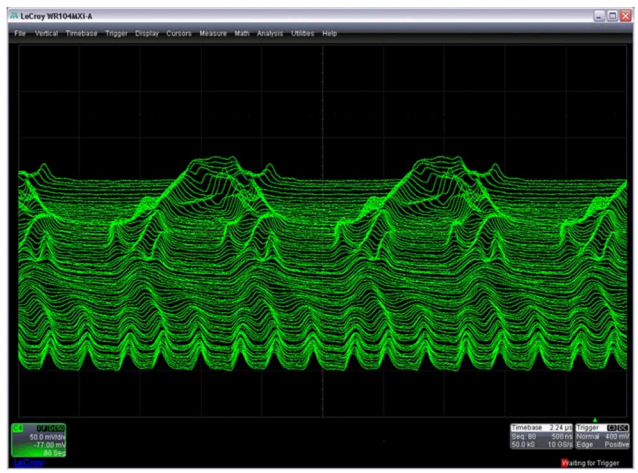

Figure 1: Real mountain range data for a bunch merge in Booster.

Some ML approaches, such as reinforcement learning (RL), do not learn machine parameters, but rather environments—making them amenable to other development paths. The importance and under-explored benefits of RL for accelerator optimization problems has already been recognized by the community [1]. We therefore pursued a framework for investigating RL optimization by replacing the accelerator, attendant diagnostics, and controls with a physics-based simulator mimicking real machine data structures and communications. Such a simulator was created for Booster and AGS using the Python programming language.

## ENVIRONMENT TO SIMULATE

Bunches merge in Booster for injection into AGS, and bunches merge in AGS for injection into RHIC. To diagnose a merge, a wall current monitor (WCM) is used. The WCM generates a voltage vs. time signal in response to passing bunches. Subsequent signal traces are stacked on an oscilloscope to create a so-called mountain range plot; see Fig. 1 for an example from Booster. A certain number of accelerator periods separate each trace, and this number can vary depending on the merge. It is typically between 100 and 200 turns.

In addition to time between acquisitions, captured mountain range data typically varies with respect to oscilloscope

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
† lnguyen@bnl.gov

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-FR2A004

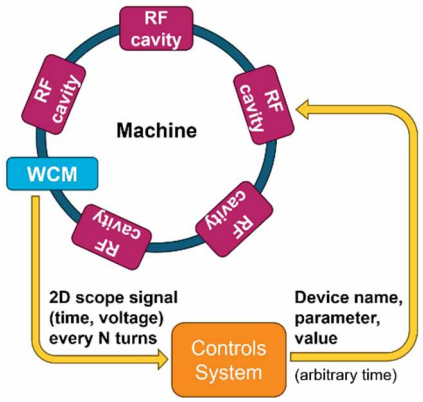


Figure 2: Cartoon representation of an accelerator with WCM, RF cavities (arbitrary number), and input/output.

settings, such as sampling rate, timebase, and trigger delay. The measurement process itself imparts noise.

Bunch merging is accomplished via RF gymnastics with different RF harmonics—but not necessarily different RF cavities. For example, this is generally the case in Booster, where only two physical cavities exist, but RF supplies are switched mid-merge to include a greater number of harmonics. Voltage and phase are the available controls for a given harmonic, and changes to these are made by specifying the device name, device parameter, and value in the Controls System. There will be some deviation between the set value and the value that appears on the cavity.

Booster and AGS naturally differ from each other in terms of RF, beam energy, slip factor, and other machine parameters. Settings for the same machine also vary for different merge patterns, species, and/or energies. However, both machines, regardless of configuration, satisfy the general concept of a bunch-merging ring diagnosed by a WCM and controlled by a prescribed communications syntax, as shown in Fig. 2.

## THE SIMULATOR

Figure 3 shows the workflow of the simulator, which in fact comprises two distinct simulated components working in concert: a physics-based simulator and a diagnostics simulator. The former operates in 2D phase space and tracks individual particle movement in response to simulated RF cavities, evolving the beam with every turn through the accelerator in accordance with the well-known longitudinal phase-space mapping equations [2]:

$$\delta_{n+1} = \delta_n + \frac{eV}{\beta^2 E}(\sin\varphi_n - \sin\varphi_s) \quad (1a)$$

$$\varphi_{n+1} = \varphi_n + 2\pi h\eta\delta_{n+1} \quad (1b)$$

The latter, meanwhile, operates in the time domain and attempts to replicate as faithfully as possible the output of a real WCM during a simulated merge. The diagnostics simulator does not itself contribute to the physics model governing the merge. The overall effect is a realistically noisy simulated environment that encapsulates a proven physics model.

### *Interaction Paradigm*

The physics-based portion of the simulator begins by taking supplied values for voltage and phase (upper right corner of Fig. 3,) to be substituted into the mapping equations [Eqs. (1a) and (1b)]; variable noise can be added to the RF cavities and/or bunches to simulate the behavior of a real machine, which can exhibit some deviation from nominal values. The physics-based simulator then passes its phase space projection to the diagnostics simulator to inherit WCM noise and timing characteristics. This voltage vs. time signal finally reaches the user or agent program for decisions regarding future voltage and/or phase values.

In this way, the two orange blocks allocated to user/RL interaction in Fig. 3 are designed to be interchangeable to cut down on time spent developing in the real environment (left side of Fig 3).

### *Programming Paradigm*

The above-described flexibility, workflow, and interaction paradigm are well suited for object-oriented programming (OOP). OOP is supported by many programming languages, but the simulator's intended purpose of facilitating ML/RL made Python the natural choice among them.

The simulator's objects are discussed in more detail in the next section.

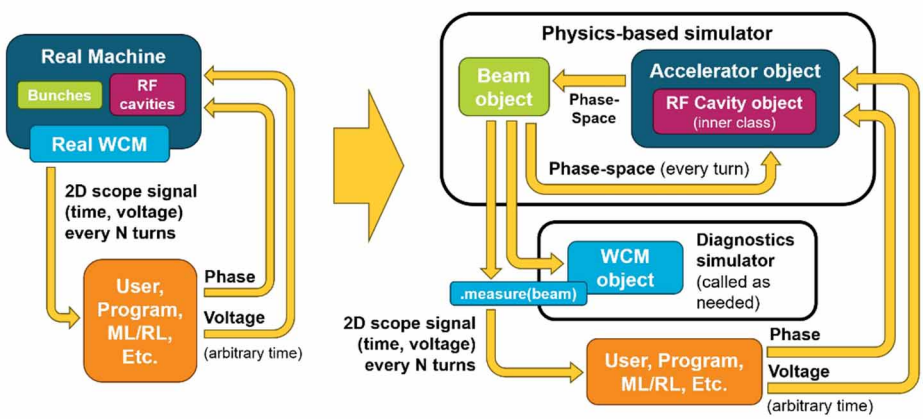


Figure 3: Bunch merge simulator workflow. The simulator is designed so that user/RL interaction (orange box) developed in the simulated environment (right) may be implemented in the real environment (left) without modification.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-FR2A004

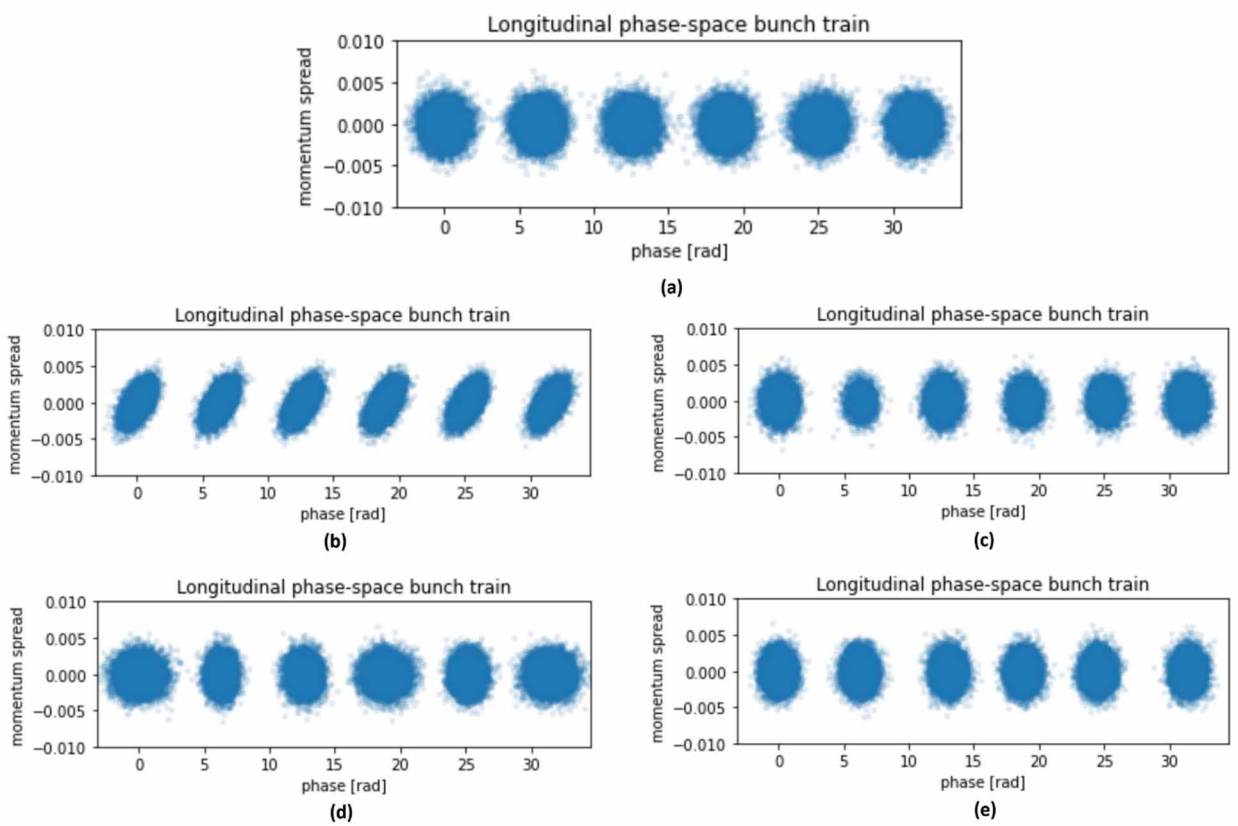


Figure 4: (a) `Beam` object with no covariance or added bunch variation (noise). The same object is also shown with (b) non-zero covariance, (c) non-zero particle number variation, (d) non-zero bunch length variation, and (e) non-zero bunch timing variation.

## CODE OVERVIEW

The simulator is handled by four objects, although one of the objects—the RF `Cavity` object—belongs to an inner class of the `Accelerator` class and is only defined as such for conceptual and organizational purposes.

### *Beam Object*

The `Beam` object contains the 2D array describing the beam in 2D phase space (momentum spread vs. phase). Figure 4(a) shows an example of a plotted `Beam` object comprising six 80-nanosecond bunches constituting one full orbit (i.e., 12π radians in phase) at 400 kHz revolution frequency, in the base configuration (i.e., no covariance or added noise). Each bunch contains 10,000 particles distributed according to a multivariate Gaussian distribution.

**Instantiation arguments** The object is created by passing the following variables to the `Beam` class: particles per bunch, total number of bunches in orbit, signal strength of the bunches on the WCM, bunch length, momentum spread, covariance, particle number variation, bunch length variation, bunch timing variation, phase resolution, and Gaussian filter sigma (σ).

**Noise parameters** The covariance and bunch variations can be combined arbitrarily to create different non-ideal beams. Figures 4(b)-(e) show the effect of these settings on the created `Beam` object. The availability of these options form part of the robustness-testing capability of the simulator.

**Phase space to time signal (beam projection)** The first step in transforming the phase-space representation of the beam into a time signal is the projection onto phase. This step is achieved numerically by generating a histogram. The phase resolution determines the number of bins used for the particle histogram, which lends a natural degree of noise. The histogram is turned into a smooth function by passing it through a Gaussian filter. Figure 5 shows the effect that the Gaussian filter sigma has on the resulting smoothed function. A larger sigma overcomes the noise incurred when a sub-optimal number of bins and/or low number of particles is used, but its effect is equivalent to ringing in the oscilloscope. The combination of these settings must therefore be chosen appropriately when replicating an environment. The resulting 2D array is stored as the attribute `phase_signal`.

From here, it is straightforward to convert phase to time by the relationship $T = 1/f$ and knowing the frequency corresponding to $2\pi$ of the RF bucket. That is, if $\varphi$ is the value along the x-axis in phase, then the value along the x-axis in time is given by $t = \varphi/\omega_{RF}$ , where $\omega_{RF} = 2\pi f_{RF}$. The y-axis is also rescaled via the signal strength parameter. The resulting 2D array is stored as the attribute `raw_time_signal`.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-FR2A004

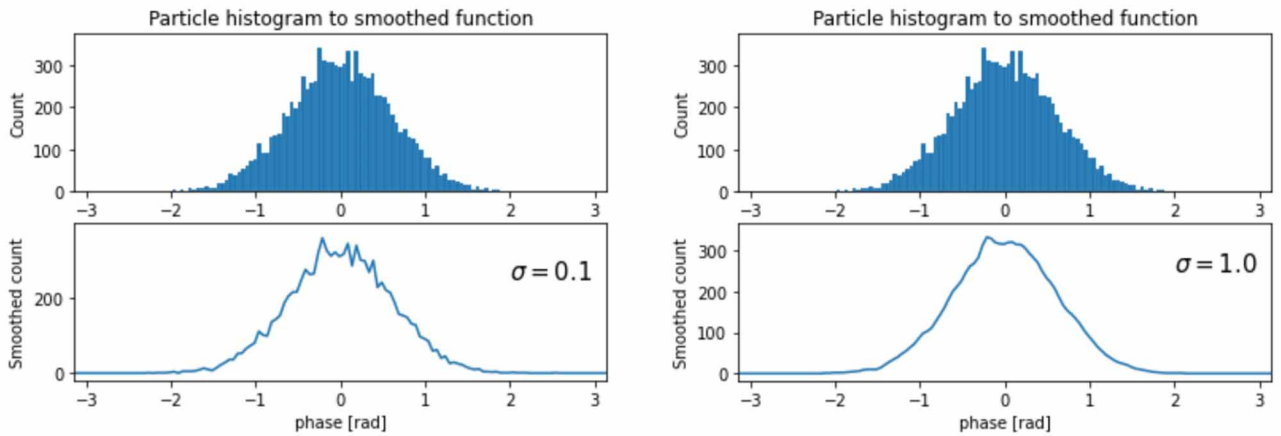


Figure 5: Phase-space projection via histogram. The particle histogram becomes a smooth function by applying a Gaussian filter. The two bottom plots differ only in the sigma value used for the filter (larger sigma means smoother function).

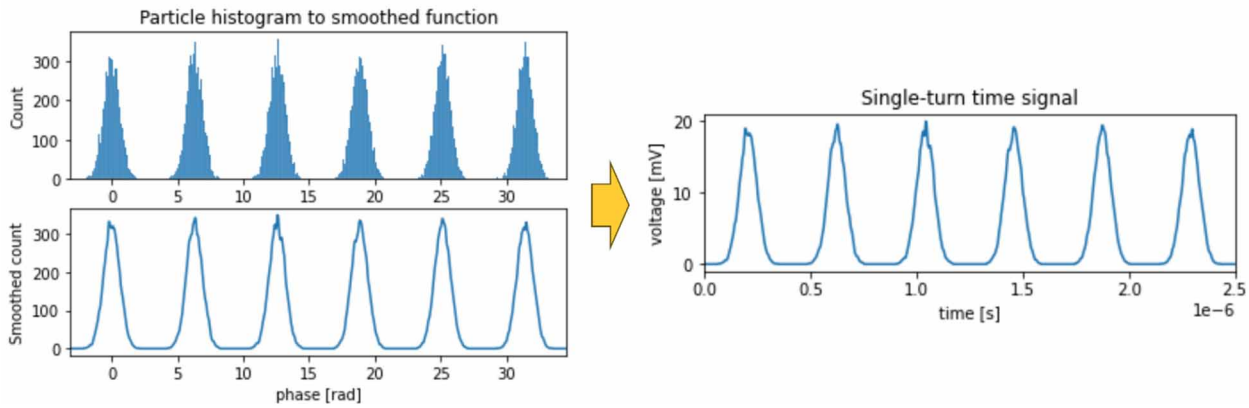


Figure 6: Converting the phase-space projection in phase to time. In this example, a 400 kHz revolution frequency is used, which equates to 2.5 µs for a full orbit and agrees with the plot on the right. The instantiation argument `signal_strength` rescales the y-axis and is here equal to 20 mV.

This entire process is handled by the `collapse()` method of the `Beam` object and is depicted in Fig. 6.

**Other methods** In addition, there is a `beamLoss(loss)` method that causes a global loss of particles and a `phaseShift(shift)` method that uniformly shifts the bunches in phase inside the RF buckets. The latter can be used to simulate a change in orbit length, for example.

### *WallCurrentMonitor Object*

The `WallCurrentMonitor` object imparts acquisition noise and other simulated scope properties to the raw time signal of the `Beam` object for a more authentic output. It is also responsible for creating the mountain range plot.

**Instantiation arguments** The object is created by passing the following variables to the `WallCurrentMonitor` class: sampling rate, number of samples per acquisition, turns between acquisitions, number of acquisitions, trigger delay, bit resolution of scope, and acquisition noise.

**Simulated measurement and the accumulator** Whenever the object method `measure(beam)` is called, Gaussian noise is added to the raw time signal of the passed `Beam` object and stored as another `Beam` attribute `measured_time_signal` (also a 2D array). See Fig. 7 for the case when rms acquisition noise is set to 0.4 mV.

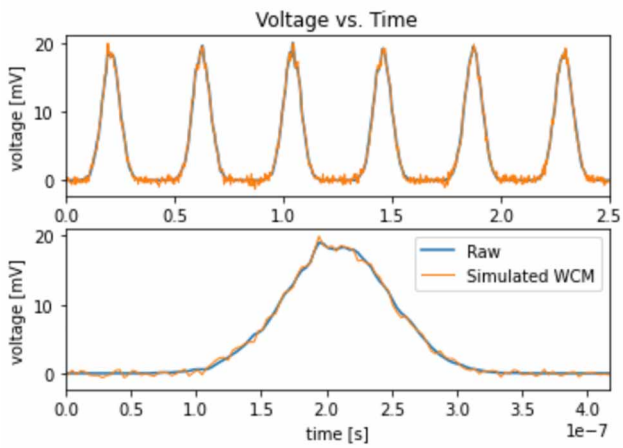


Figure 7: The effect of the `measure(beam)` method of a `WallCurrentMonitor` object. The bottom plot is a closeup of the first bunch in the top plot.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-FR2A004

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

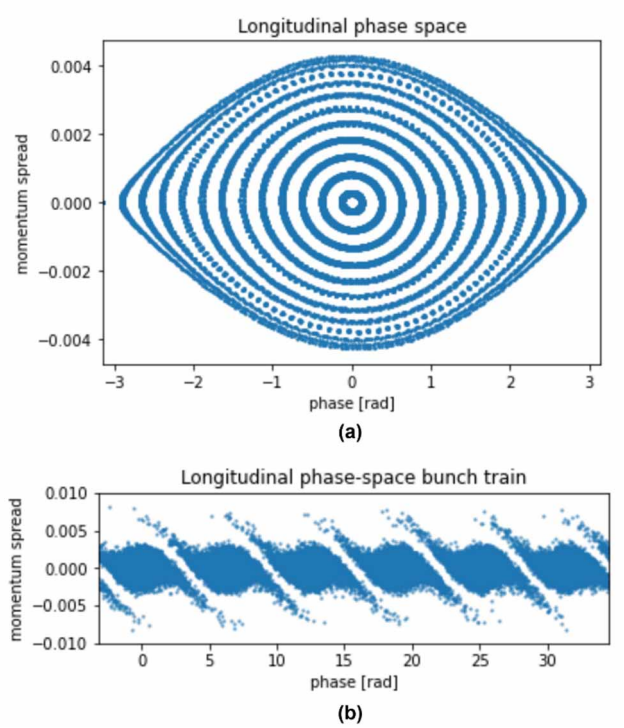


Figure 8: (a) Expected constant phase-space ellipses for on-momentum particles in a constant RF environment. (b) Phase wrapping for full orbit tracking.

Every simulated measurement is stacked in the `WallCurrentMonitor` attribute `accumulator`. This attribute can be called if an RL policy would like to take history into account.

**Mountain range plot** The `display()` method plots all the data stored in the accumulator for the appearance of a genuine mountain range plot. Example plots are shown in the Results section.

**Other methods** In addition to `measure(beam)` and `display()`, there is a `clear()` method for restarting the accumulator for a new WCM display.

### *Accelerator and RF Cavity Objects*

The physics simulation engine is found in the `Accelerator` object. The difference between a Booster simulator and an AGS simulator is determined here.

**Instantiation arguments** The `Accelerator` object and its child `Cavity` object are created simultaneously by passing the following variables to the `Accelerator` class: species, particle rest energy, particle charge number, machine name, machine energy, slip factor, revolution frequency, merge harmonics, initial RF voltages, voltage device/parameter names, initial RF phases, phase device/parameter names, voltage noise, and phase noise.

**Harmonics** All harmonics used in the merge are used to instantiate the `Accelerator` object and are implicitly given their own cavity in simulation space. However, since a lump-element model is used for simplification, a more accurate description might be that there exists one cavity capable of sustaining all harmonics without distortion.

**Phase-space mapping equations** Equations (1a) and (1b) must be modified to handle a superposition of RF fields. The equations used instead are

$$\delta_{n+1} = \delta_n + \sum_i^{len(h)} \frac{ZeV_i}{\beta^2 E}\left(sin\,\varphi_{n,i} - sin\,\varphi_{s,i}\right) \quad (2a)$$

$$\varphi_{n+1} = \varphi_n + 2\pi h_N \eta \delta_{n+1} \quad (2b)$$

where $i$ corresponds to matching indices in the `Cavity.harmonics`, `Cavity.voltages`, and `Cavity.phases` arrays, and $h_N$ is the base harmonic used for preserving scale. With the lump-element simplification, $\varphi_s$ for each harmonic can be approximated by $\varphi_i$. All terms are always present in the summation in the code; harmonics not participating in the merge for a given iteration simply have their voltage set to zero.

Figure 8(a) shows the expected phase-space ellipses for on-momentum particles distributed in phase when iterating through the simulator with constant RF conditions. Figure 8(b) demonstrates the phase-wrapping required for full orbit tracking. These investigations were performed first to prove the soundness of the simulator's baseline behavior.

**Cavity noise** If enabled, voltage and cavity noise is inserted when the phase-space mapping equations are applied. Nominal values are stored as attributes.

**Attributes and methods** `Accelerator` and `Cavity` object attributes and methods central to the simulation engine are discussed in the next sections. In addition to those, there are `unloadCommands()`, `reset(attribute)`, `getValue(parameter)`, and `setValue(parameter,value)` methods.

```
v_names = [['Cavity1', 'h1_VoltageSetpoint'],
           ['Cavity2', 'h2_VoltageSetpoint'],
           ['Cavity3', 'h3_VoltageSetpoint'],
           ['Cavity4', 'h6_VoltageSetpoint']]

ph_names = [['RFSupply1', 'h1_PhaseOffset'],
            ['RFSupply2', 'h2_PhaseOffset'],
            ['RFSupply3', 'h3_PhaseOffset'],
            ['RFSupply4', 'h6_PhaseOffset']]

# Specify each command as a row in the following format:
# [Time in ms, ['Device name', 'Parameter'],value]
# Voltage in kV, phase in rads

merge_commands = [
    [0, ['Cavity4', 'h6_VoltageSetpoint'], 2.5],
    [8, ['Cavity4', 'h6_VoltageSetpoint'], 1.25],
    [8, ['Cavity3', 'h3_VoltageSetpoint'], 0.8335],
    [16, ['Cavity4', 'h6_VoltageSetpoint'], 0],
    [16, ['Cavity3', 'h3_VoltageSetpoint'], 1.25],
    ...
    [48, ['Cavity3', 'h3_VoltageSetpoint'], 0.35],
    [48, ['Cavity1', 'h1_VoltageSetpoint'], 1.46045],
    [60, ['Cavity3', 'h3_VoltageSetpoint'], 0],
    [60, ['Cavity2', 'h2_VoltageSetpoint'], 0.730225],
    [72, ['Cavity2', 'h2_VoltageSetpoint'], 0]
    ]
```

Figure 9: Controls system names and merge program used for simulation. The simulator contains no built-in names but rather receives them from the user.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-FR2A004

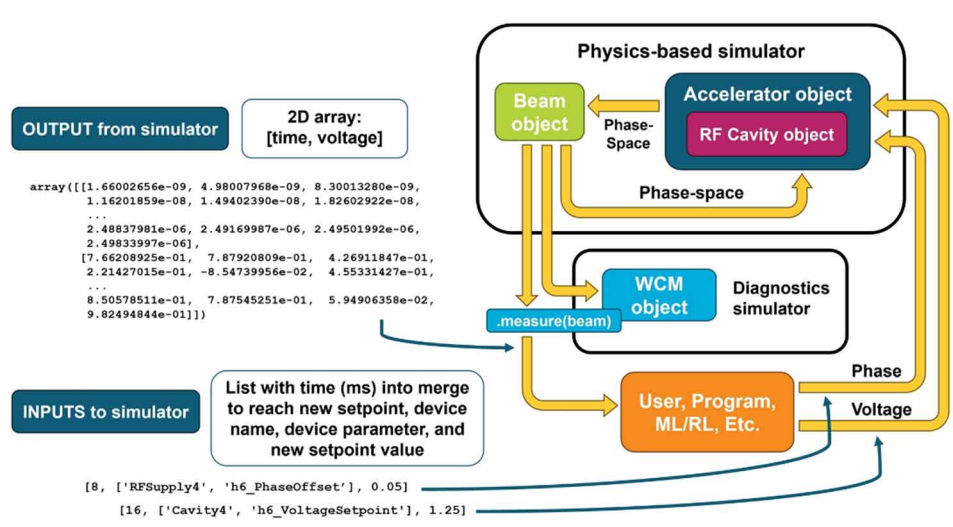


Figure 10: Summary of data structure.

## CONTROLS ARCHITECTURE

When the `Accelerator` object is created, its class is passed two lists of controls system names: one with the device name and parameter corresponding to the voltage control of a given harmonic and another for the phase control. The syntax for each element in the list is therefore another list of the form

```
['Device name', 'Device parameter']
```

The instantiation process packages both lists as the attribute `Cavity.params`. Whenever a search is performed on `params` (as when a command is sent), the result is mapped to the appropriate element in `voltages` or `phases`, which are themselves mapped to a value in `harmonics`. In this way, the simulator possesses no built-in names but rather accepts whatever names are specific to the controls system of the simulated environment.

When creating the merge program, the same syntax is used for specifying the control; it is sent together with the new setpoint and the time into the merge at which the new setpoint must be achieved. Any changes to `voltages` and/or `phases` are used in subsequent iterations.

Figure 9 shows the dummy controls system names used in the example simulation, followed by an abbreviated version of the merge commands, to illustrate the controls architecture. The merge commands are lifted from Ref. [3], albeit with a reduction in the time values by a factor of 10 to achieve an overall 72-millisecond merge, rather than a 720-millisecond merge. Figure 10 summarizes the data structure occurring at the input and output of the simulator once all the above code is implemented.

## SIMULATION AND RESULTS

The code for a user to run the simulator is provided in Fig. 11. The advantages of OOP and the chosen class design are clear from the conciseness and readability of the code. Instantiation arguments are initialized in an initialization block where units and variable formats are documented. Following instantiation, the merge program is loaded via the `loadCommands(commands)` method of the `Accelerator` object (here named `machine`). The `WallCurrentMonitor` object (here named `wcm`) invokes its `measure(beam)` method to take the first simulated measurement and thus begin a WCM display stack. The actual simulation block consists of only three lines: the `for` loop dictating the number of WCM measurements, a call to the `simulate(beam, turns)` method of the `Accelerator` object [`turns` is the number of steps through Eqs. (2a) and (2b)], and another measurement. Lastly, the simulated mountain range plot is displayed.

Inside the `loadCommands` method, the time in the command line is converted to turn number based on the revolution frequency. Unique turn numbers populate an `action` list. It is important to remember that the RF commands are not completed instantaneously at the specified time; rather, values are ramped up or down to achieve the next setpoint at the specified time. Thus, for every element in `action`, there is a row in `Cavity` attributes `v_increment` and `ph_increment`

```
wcm = WallCurrentMonitor(N_turns,N_samples,N_length,
                         wcm_noise,trigger_delay,bit_res)

beam = Beam(N_bunches,N_particles,cov,bunch_amp_noise,
            sigma_noise,sigma_ph,phase_jitter,
            signal_strength,b_num,gauss_sigma)

machine = Accelerator(machine_name,species,E_0,Z,E,eta,
                      h_list,v_list,ph_list,v_names,
                      ph_names,v_noise,rf_ph_noise)

machine.loadCommands(merge_commands)

wcm.measure(beam)  # Log initial beam

for i in range(N_measurements):
    machine.simulate(beam, wcm.turns)
    wcm.measure(beam)

wcm.display()  # Mountain range plot
```

Figure 11: Simulator code as implemented by user.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-FR2A004

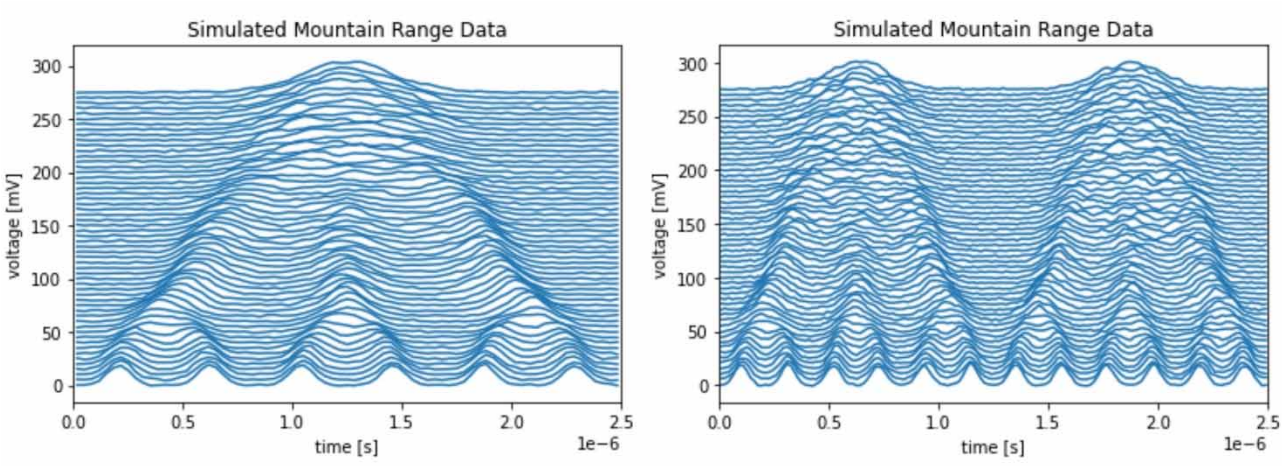


Figure 12: Simulation results using a known Booster merge program.

containing the incremental voltage value and incremental phase value, respectively, for each harmonic to reach the next setpoint as specified. If a control is not involved in the ramp for that turn, its incremental value is obviously zero. Note that these incremental values can be used to constrain merges to realistic ramps.

Once inside the `simulate` method, Eqs. (2a) and (2b) are applied once per iteration, signaling one turn. Before proceeding to the next iteration, `v_increment` and `ph_increment` are added to `voltages` and `phases`, respectively, so that subsequent iterations are responding to the correct RF conditions. The running `turn` and `line` attributes of the `Accelerator` object keep track of the position within the `action` list and incremental value arrays.

Figure 12 shows the results of the simulation (i.e., output of `display` method) for two different initial bunch trains. A surprisingly small number of particles achieved these results: only 2,400 distributed equally among the initial bunches. This shows the benefit of employing a Gaussian filter in the projection step. Approximately 28,000 turns were simulated, with a total compute time of about 20 minutes when performed on a CPU utilizing one core. Although mountain range data for an identical merge could not be found, the real 3-to-1 Booster merge shown in Fig. 13 is illustrative of the success of the simulator.

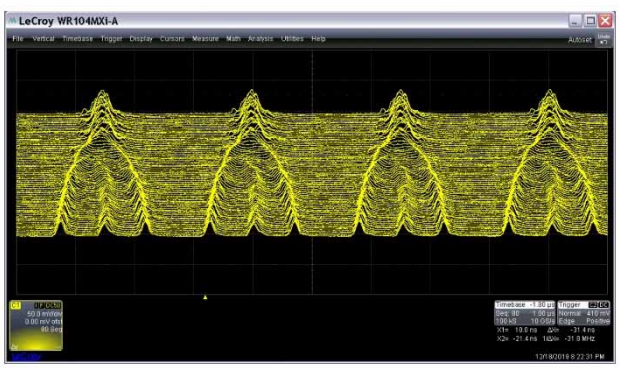


Figure 13: Real mountain range data for a 3-to-1 Booster merge for comparison.

## CONCLUSION

We have created a physics-based simulator in Python that mimics our bunch merge environment and diagnostics by combining longitudinal phase-space mapping and phase-space projection for time signal replication. A diagnostics simulator is included in the framework. The resulting code used to run the simulator by a user is minimal and comprehensible thanks to OOP and careful design.

Due to its authentic data and controls structure, as well as the options for injecting noise, the simulator can be used for RL development for improving bunch merges. Other envisaged uses include operator training, machine troubleshooting, and other systems investigations.

Regarding future work, the simulator will continue to be expanded beyond the presented capabilities, including the option to accept console input rather than use a loaded merge program. This is expected to improve RL development/performance since the algorithm would not need to wait until the very end of the merge for feedback. It is also likely that the `simulate` method (or a version of it) will be rewritten to perform the computationally intensive phase-space mapping portion in C for full-scale simulations (>>10,000 particles per bunch) when GPUs are insufficient/unavailable.

## REFERENCES

[1] K. Brown and S. Biedron, "Summary of the 3rd ICFA Beam Dynamics Mini-Workshop on Machine Learning Applications for Particle Accelerators", in *Proc. IPAC'23*, Venice, Italy, May 2023, pp. 4440-4443. doi:10.18429/jacow-ipac2023-thpl010

[2] S. Y. Lee, *Accelerator Physics*, Singapore, World Scientific Publishing, 2019.

[3] C. J. Gardner, "Preservation of the distribution of beam particles with respect to longitudinal oscillation amplitude in a 3 to 1 bunch merge," Brookhaven National Laboratory, Upton, NY, USA, Rep. BNL-212078-2019-TECH C-A/AP/625, Sept. 2019.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI