

Lawrence Berkeley National Laboratory

LBL Publications

Title

The Memory Scaling of Reverse-Mode Differentiation in Particle Accelerator Simulations with Space Charge

Permalink

<https://escholarship.org/uc/item/23c2k8dq>

Journal

Pasc 2026 Platform for Advanced Scientific Computing Conference

Authors

Dhamrait, Arjun

Zoni, Edoardo

Huebl, Axel

et al.

Publication Date

2026-06-29

DOI

10.1145/3815572.3815749

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed

The Memory Scaling of Reverse-Mode Differentiation in Particle Accelerator Simulations with Space Charge

Arjun Dhamrait

Edoardo Zoni

Axel Huebl

Ji Qiang

Chad Mitchell

Lawrence Berkeley National

Laboratory

Berkeley, California, USA

Ryan Roussel

SLAC National Laboratory

Stanford, California, USA

Jan Kaiser

Deutsches Elektronen-Synchrotron

DESY

Hamburg, Germany

Chenran Xu

Argonne National Laboratory

Lemont, Illinois, USA

Jean-Luc Vay

Remi Lehe

Lawrence Berkeley National

Laboratory

Berkeley, California, USA

Abstract

The recent development of *differentiable* simulation codes for particle accelerators has enabled gradient-based workflows that promise finer control and more realistic modeling of accelerator facilities. However, when using reverse-mode automatic differentiation, the memory usage continuously increases during the simulation, and can potentially exceed the available hardware memory – especially when costly space charge computation is included. To study the memory requirements for differentiable simulations, we have implemented space charge in Cheetah, a PyTorch-based beam tracking code that supports reverse-mode differentiation. We find that the memory usage for reverse-mode differentiation grows linearly with the number of macroparticles and cells, and that it is proportional to the number of space charge kicks involved in the simulation. This general scaling can be used to evaluate whether a given differentiable simulation is feasible given hardware memory constraints.

CCS Concepts

• **Applied computing** → **Physics**; • **Computing methodologies** → **Simulation types and techniques**.

Keywords

automatic differentiation, reverse-mode differentiation, particle tracking, particle-in-cell, particle accelerators

ACM Reference Format:

Arjun Dhamrait, Edoardo Zoni, Axel Huebl, Ji Qiang, Chad Mitchell, Ryan Roussel, Jan Kaiser, Chenran Xu, Jean-Luc Vay, and Remi Lehe. 2026. The Memory Scaling of Reverse-Mode Differentiation in Particle Accelerator

Simulations with Space Charge. In *Platform for Advanced Scientific Computing Conference (PASC '26)*, June 29–July 01, 2026, Bern, Switzerland. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3815572.3815749>

1 Introduction

Modeling beam dynamics in particle accelerators is key to both the design and operation of accelerator facilities. Accordingly, a number of beam tracking simulation codes have been developed over the years – including Bmad, Elegant, Impact, ImpactX, MAD-X, Ocelot, SixTrack, Xsuite, just to name a few – and are routinely used at various facilities. These codes track individual beam particles traveling through a series of accelerator lattice elements (quadrupoles, steering magnets, RF cavities, etc.) and many of these codes also incorporate the effect of 3D space charge. Although incorporating space charge significantly increases computational costs, it is crucial for capturing phenomena such as intensity-dependent beam losses [5], space-charge-driven emittance growth [27], as well as certain types of collective beam instabilities [15, 28].

In recent years, several *differentiable* beam tracking codes have been developed [6, 10, 11, 16, 25]. Beyond computing output quantities (e.g., final beam size) from given inputs (e.g., quadrupole strengths, initial beam parameters, etc.), *differentiable* tracking codes can also calculate the derivatives of those outputs with respect to the inputs. As such, differentiable tracking codes have enabled a number of new workflows, including gradient-based accelerator lattice optimization [10], gradient-based identification of unmeasured lattice misalignments [10], and more efficient variants of Bayesian optimization [2, 10]. These differentiable codes also enable training neural networks *while they are embedded* within a broader simulation pipeline, as is the case in generative phase space reconstruction [9, 22, 23]. Furthermore, they can also be used to study the stability of dynamical systems [17, 20]. Finally, differentiable codes could also be leveraged for more efficient *Simulation-Based Inference* (SBI) [3].

The abovementioned differentiable tracking codes make use of *automatic differentiation* – a technique that enables calculation of



This work is licensed under a Creative Commons Attribution 4.0 International License. *PASC '26, Bern, Switzerland*

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2734-4/2026/06

<https://doi.org/10.1145/3815572.3815749>

the derivatives to machine precision (as opposed to approximate numerical methods for derivatives, such finite difference) without requiring developers to manually implement derivative calculations in the source code. Automatic differentiation itself can be performed using either the *forward-mode* or *reverse-mode* technique [4, 7]. In *forward-mode* differentiation, each variable a is paired with its gradient with respect to the inputs ($\partial a / \partial x_i$ for each input x_i), and each arithmetic operation on this variable (e.g., addition, multiplication) is overloaded to simultaneously carry out the corresponding chain-rule operation for its gradients. As a result, the required computational resources (both in time-to-solution and memory usage) scale linearly with the number of input variables.

By contrast, in *reverse-mode* differentiation, the computation of derivatives is carried out in two distinct “passes”. In the “forward pass”, the simulation code is run using ordinary arithmetic operations to compute the output f , while recording data required for later gradient calculations. Then in the “backward pass”, each operation is revisited in reverse order – from output f to the inputs x_i – going through each intermediate variable a and computing $\partial f / \partial a$ until reaching $\partial f / \partial x_i$ [4, 7] (see App. A for an illustration based on a simple example). As a result, the scaling of time-to-solution is largely independent of the number of input variables, making the reverse mode faster than the forward mode when handling a large number of inputs. This is especially valuable when training neural networks while they are embedded in a simulation [9, 22, 23], a task that requires finding gradients with respect to thousands or even millions of inputs (the weights of the neural network). However, one potential issue of reverse-mode differentiation is its memory usage. The memory usage indeed increases steadily during the forward pass, because of the need to continuously record data that will be used during the backward pass. As a result, there is a possibility that the simulation may run out of memory before the forward pass is completed.

The aim of this paper is thus to understand the scaling of memory usage in reverse-mode differentiation, in the particular case of beam tracking simulation with space charge. We study this scaling with respect to the number of macroparticles in the simulation, and the number of grid cells used for the space charge calculation. In order to carry out this study, we added support for space charge in the code Cheetah, an open-source beam tracking code that leverages PyTorch’s built-in support for reverse-mode differentiation. (For more details on the Cheetah code itself, see [10].) To our knowledge, this is the first open-source beam tracking code which supports space charge with reverse-mode differentiation. The space-charge algorithm implemented in Cheetah uses standard Particle-In-Cell methods and is detailed in Sec. 3.2; benchmarks that confirm its accuracy are summarized in App. B.

While we measure the memory scaling specifically for the code Cheetah, the big O scaling (e.g., whether the memory increases linearly or quadratically with the number of macroparticles) should not be specific to Cheetah and should apply to any beam tracking code with reverse-mode differentiation. Ultimately, these results can serve to estimate the memory requirements for a given differentiable beam tracking simulation, and whether this simulation is feasible on a given piece of computing hardware.

2 Structure of beam tracking codes with space charge, and memory implications

To gain a clearer understanding of how memory usage scales, it is helpful to first review the basic operating principles of beam tracking codes. These codes simulate the motion of N_{part} macroparticles of a charged particle beam – each often representing a large number of physical particles – through a sequence of N_{elements} lattice elements. Importantly, rather than parameterizing trajectories using time t , these simulations typically employ the path length s along the accelerator lattice as the independent variable. As the macroparticles advance in s , the algorithm updates their coordinates, which include the transverse positions x and y , the relative arrival time t , the transverse momenta p_x , p_y , and the energy deviation $\delta E = E - E_0$ of each macroparticle – where E_0 is the energy along the designed *reference* trajectory for the accelerator lattice. (Depending on the specifics of each code, normalized versions of these coordinates may be used in practice.)

In cases where space-charge is neglected, each macroparticle is evolved *independently* through the lattice elements. This uses a *map* that gives the macroparticle coordinates at the exit of the element (at s_{exit}) as function of those at the entrance (at s_{entrance}). For simple enough representations of the lattice elements, the map often consists in a known, closed-form analytical expression.

On the other hand, when space-charge is taken into account, there is no such analytical expression. A common approach is to employ second-order Strang splitting (with higher-order variants also in use). In this scheme, each lattice element is divided into N_{slices} segments of length Δs . For each slice, particles are first advanced without space-charge over $\Delta s/2$ (independently, using maps), then their canonical momenta are updated to account for the space-charge interaction over Δs at fixed x, y, t , and finally they are advanced again without space-charge for the remaining $\Delta s/2$. The effect of space-charge over Δs is commonly referred to as a “space charge kick”. This is typically calculated using the Particle-In-Cell (PIC) method: macroparticles deposit their charge density onto a spatial grid, the fields are then computed on that grid by solving Poisson’s equation, and the resulting Lorentz forces are interpolated back to the macroparticle positions to update their momenta. Further details are provided in Sec. 3.2.

In order to illustrate the typical behavior of memory usage in these simulations, we consider an electron bunch traveling through a drift (i.e., free space propagation) and expanding under its own space-charge. This setup is similar to the test labeled “Free Expansion of a Cold Uniform Density Bunch” in [13], and the initial electron bunch is a uniform ellipsoid defined by a transverse radius R_i and length L_i (such that particles uniformly fill the space defined by $(x^2 + y^2)/R_i^2 + z^2/L_i^2 < 1$). The drift is subdivided into $N_{\text{slices}} = 3$ slices to take into account the effect of space-charge; the physical and numerical parameters of the simulation are summarized in Table 1. The simulation is run with the code Cheetah [10] on an 80GB NVIDIA A100 GPU on the NERSC Perlmutter system. Fig. 1 shows a timeline of memory usage, when the code is run without tracking derivatives (left panel in Fig. 1), or with derivatives (right panel in Fig. 1) – in which case we use reverse-mode differentiation to compute the derivatives of the final RMS beam size $\sigma_{x,f}$ with respect to the initial beam radius $\partial \sigma_{x,f} / \partial R_i$, the initial beam length $\partial \sigma_{x,f} / \partial L_i$,

Parameter	Value
Beam charge Q	10 nC
Beam energy $\mathcal{E}$	250 MeV
Beam initial radius R_i	1 mm
Beam initial length L_i	1 mm
Length of the drift L	5.5 m
Number of macroparticles $N_{\text{part.}}$	1,000
Number of grid cells N_{cells}	32^3
Numer of slices N_{slices}	3

Table 1: Physical and numerical parameters of the simulation

the beam charge $\partial\sigma_{x,f}/\partial Q$, and the beam energy $\partial\sigma_{x,f}/\partial\mathcal{E}$. All four derivatives are calculated in a single backward pass. The details of how memory usage was measured are summarized in App. C.

Figure 1 illustrates the difference in memory behavior with and without derivative tracking. In the non-differentiable case (left panel), memory is allocated and freed during each space charge kick, with no net growth over time. Closer inspection – not shown here – indicates that, in this particular simulation, most of this transient allocation is related to computing temporary grid-based quantities used in the space-charge calculation.

In contrast, when reverse-mode differentiation is enabled (right panel), memory usage increases steadily during the forward pass and then decreases during the backward pass. As noted in the introduction and illustrated in App. A, this behavior is expected: the code continuously records necessary data during the forward pass, which is then used to compute derivatives during the backward pass.

The figure further shows that, in the forward pass, memory usage increases by the same increment during each space charge kick (i.e., within each slice). This is again expected, as each space charge kick executes the same sequence of operations, leading to the same amount of data being stored for the backward pass. Consequently, the peak memory usage (i.e., the memory usage at the end of the forward pass) should scale proportionally to N_{kicks} , the number of space charge kicks applied throughout a simulation. (When the simulation contains several lattice elements, N_{kicks} is sum of the number of slices for each element.) We expect that this scaling behavior is not unique to Cheetah and that it should apply generally to any beam-tracking code implementing reverse-mode differentiation.

Although the scaling with N_{kicks} is straightforward, the scaling with the number of macroparticles $N_{\text{part.}}$ and grid cells N_{cells} is less apparent. Yet understanding this scaling is essential for estimating whether a simulation will exceed available memory by the end of the forward pass. This topic is examined in detail in Sec. 3.

Before turning to that analysis, it is useful to make a few additional observations in Fig. 1. First of all, the evaluation of space charge kicks accounts for most of the computational cost, both for the simulation without derivatives (left panel) and for the forward pass of the differentiable simulation (right panel). The small gaps between the dashed lines for consecutive kicks correspond to the application of linear maps to each particle (as required in the Strang splitting scheme) and take negligible time. These small gaps also

do not incur any perceptible change in memory consumption, compared to the space charge kicks, and this is the reason why this paper focuses on the memory usage due to the space charge kick, rather than that due to the application of linear maps. Secondly, the forward pass in the differentiable simulation (right panel) is only marginally slower than the non-differentiable run. This is indeed one of the advantages of reverse-mode differentiation, compared to forward-mode differentiation.

3 Detailed memory scaling of the space charge kick with number of macroparticles and cells

3.1 Overall memory scaling

In order to study memory usage during the forward pass with respect to $N_{\text{part.}}$ and N_{cells} , we now run a *single* space charge kick. We use the same physical parameters as in Table 1, but we systematically vary $N_{\text{part.}}$ and N_{cells} . The increase in memory usage over one single space charge kick is measured as described in App. C.

The result of this scan over $N_{\text{part.}}$ and N_{cells} is displayed in Fig. 2, where it is compared with a linear fit of the form

$$\Delta M = \alpha_{\text{part.}} N_{\text{part.}} + \alpha_{\text{cells}} N_{\text{cells}} \quad (1)$$

where ΔM is the increase in memory usage (in bytes) after a single space charge kick – during the forward pass, with reverse-mode differentiation turned on. (See App. D for more details on how this linear fit was performed.) The corresponding values of the coefficients $\alpha_{\text{part.}}$ and α_{cells} are represented in Fig. 3 (red bars).

As can be seen in Fig. 2, the data points align with this linear fit, and thus the memory usage is linear in both $N_{\text{part.}}$ and N_{cells} . Indeed if there was any significant nonlinear term such as N_{cells}^2 or $N_{\text{part.}}^2$, we would see a clear departure of the data points from the linear fits in Fig. 2. Similarly, if there was any significant nonlinearity of the form $N_{\text{cells}} \times N_{\text{part.}}$, the data points would align along lines that would have different *slopes* for different values of N_{cells} on the top panel of Fig. 2 and for different values of $N_{\text{part.}}$ on the bottom panel of Fig. 2, which is not the case.

The fact that the memory scaling is linear is encouraging for the practical applicability of differentiable beam tracking code. Any nonlinear term (e.g., N_{cells}^2 , $N_{\text{part.}}^2$, $N_{\text{cells}} \times N_{\text{part.}}$) could indeed have led to prohibitively large memory usage, given the typically large values of N_{cells} and $N_{\text{part.}}$. For instance, it is not uncommon to use on the order of $10^5 - 10^6$ macroparticles, and similar numbers for the number of cells. Note that, one could have *a priori* expected such nonlinear terms to exist. For instance, a space charge kick involves computing the potential ϕ from the charge density ρ , on a grid (see Sec. 3.2). Since this involves solving an elliptic equation, in principle $\partial\phi_{ijk}/\partial\rho_{i'j'k'}$ is non-zero for any pair of cells $(ijk, i'j'k')$. One could have expected that this matrix of size $N_{\text{cells}} \times N_{\text{cells}}$ would need to be stored during the forward pass, so as to propagate gradients during the backward pass.

In order to better understand the observed linear scaling, in the rest of this section we break down the space charge kick into its sub-steps (Sec. 3.2) and inspect the memory scaling of some key sub-steps (Sec. 3.3, 3.4, 3.5).

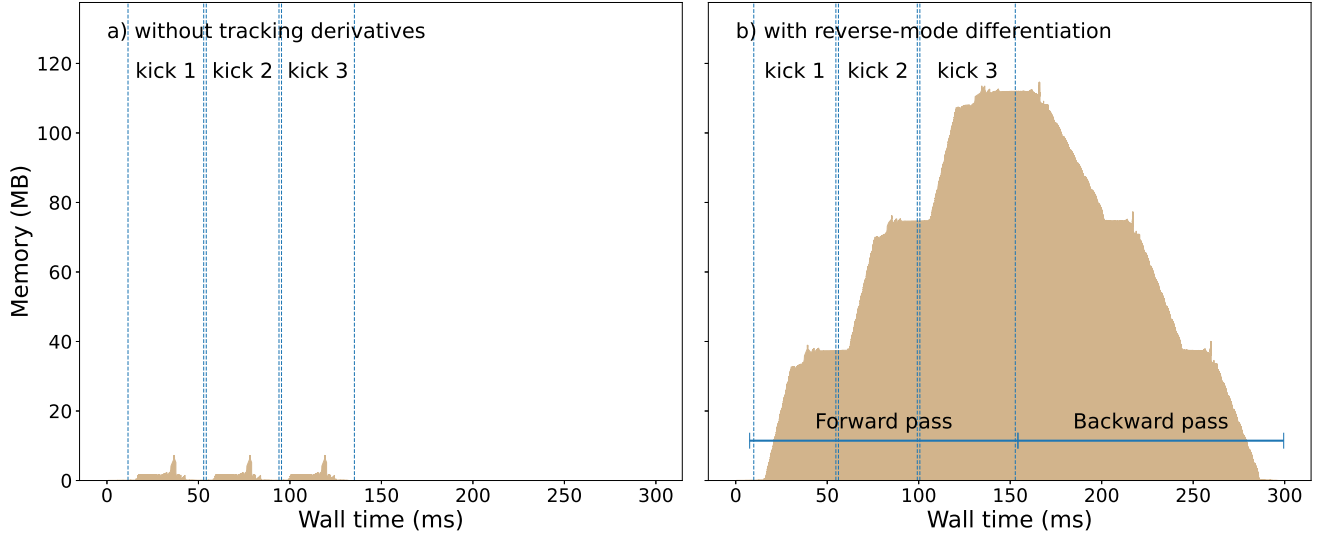


Figure 1: Timeline of memory usage for a Cheetah simulation (a) without tracking derivatives (left panel) and (b) with reverse-mode differentiation (right panel).

3.2 Breakdown of the algorithmic steps within one space charge kick

As mentioned previously, the purpose of the space charge kick is to update the momentum of each macroparticle to capture the impact of space charge over a distance Δs . In Cheetah, the space charge kick consists of the following steps, which are similar to the algorithm used in other beam tracking codes:

- **s-to-t transform:** The macroparticles are converted from a representation at fixed s (where each macroparticle is represented with $x, y, t, p_x, p_y, \delta E$) to a representation at fixed time (where each macroparticle is represented with $x, y, z = -\beta ct, p_x, p_y, p_z = \sqrt{(E_0 + \delta E)^2/c^2 - p_x^2 - p_y^2 - m^2 c^2}$).
- **Charge deposition:** A Cartesian grid that overlaps with the beam is created (with cell size $\Delta x, \Delta y, \Delta z$), and the charge density ρ is computed on the grid using the Cloud-In-Cell method [1], i.e. for each grid point indexed by ijk ,

$$\rho_{ijk} = \frac{1}{\Delta x \Delta y \Delta z} \sum_{p=1}^{N_{part.}} q_p S(\mathbf{x}_{ijk} - \mathbf{x}_p)$$

where q_p and $\mathbf{x}_p$ is the charge and position of macroparticle p , $\mathbf{x}_{ijk}$ is the position of grid point ijk , and S is the product of first-order spline functions in x, y and z [1].

- **Field solver:** For a relativistic beam propagating along the z axis with a Lorentz factor γ , the Lorentz force $\mathbf{F} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B})$ can be obtained with $\mathbf{F} = -q\nabla\phi/\gamma^2$ where ϕ is found by solving a modified Poisson equation [24]

$$\partial_x^2 \phi + \partial_y^2 \phi + \frac{1}{\gamma^2} \partial_z^2 \phi = -\frac{\rho}{\epsilon_0}. \quad (2)$$

Cheetah solves this Poisson equation by using the Integrated Green Function (IGF) method [18] to obtain ϕ on the grid,

and then computes $\mathbf{F} = -q\nabla\phi/\gamma^2$ on the same grid points using centered finite differences.

- **Field gather:** The Lorentz force $\mathbf{F}$ is interpolated from the grid points to the macroparticles using:

$$\mathbf{F}(\mathbf{x}_p) = \sum_{ijk} S(\mathbf{x}_p - \mathbf{x}_{ijk}) \mathbf{F}_{ijk}$$

- **Momentum update:** The momentum of each macroparticle $\mathbf{p} = (p_x, p_y, p_z)$ is updated using

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{F} \Delta t$$

where $\mathbf{F}$ is the interpolated Lorentz force and $\Delta t = \Delta s/\beta_z c$ is the amount of time it takes the beam to travel the distance Δs .

- **t-to-s transform:** The energy deviation δE is obtained with $\delta E = c \sqrt{p_x^2 + p_y^2 + p_z^2 + m^2 c^2} - E_0$.

Note that the approach of using the modified Poisson equation Eq. (2) is mathematically equivalent to solving the regular Poisson equation in the beam's rest frame, and transforming the Lorentz force back to the laboratory frame – which is the approach used in some of the other beam tracking codes.

The t -to- z transform, z -to- t transform and momentum update correspond to a series of element-wise operations on macroparticle arrays of size $N_{part.}$ (element-wise multiplications, additions, square root), and thus it seems natural that the memory increase incurred during these parts of the forward pass scales as $O(N_{part.})$. (Again see App. A for an example of why memory increases during the forward pass.) On the other hand, the scaling of the charge deposition, field solver and field gather is less intuitive and will be examined in more detail in the following subsections.

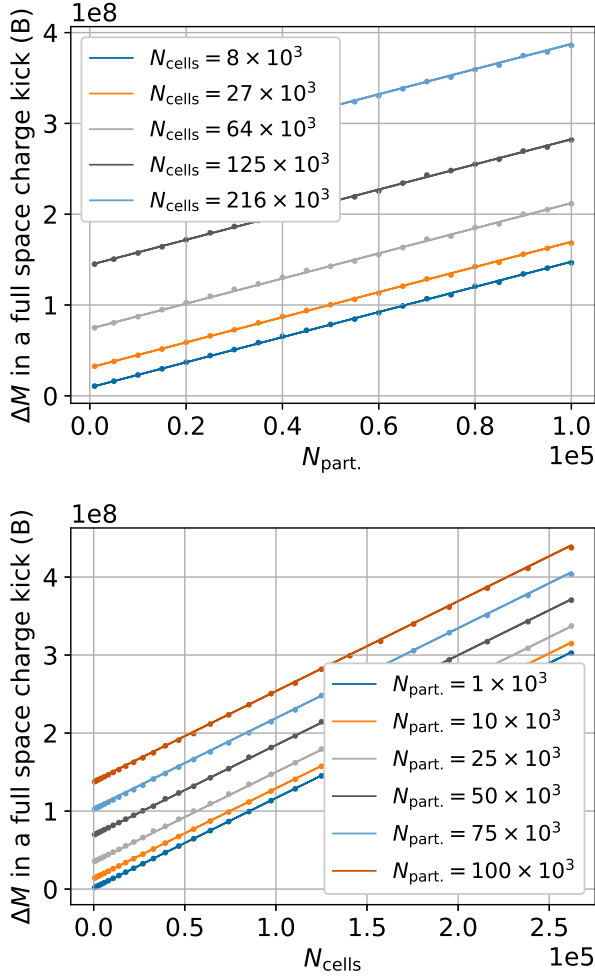


Figure 2: Total change in memory usage over one space charge kick (forward pass, with reverse-mode differentiation turned on), as a function of the number of macroparticles and cells. Eq. (1) is fitted to the data, and the corresponding coefficients are represented in Fig. 3. The lines show the evaluation of this fitted formula for fixed number of cells (top) and fixed number of macroparticles (bottom).

3.3 Memory scaling of the field solver

The field solver uses the Integrated Green Function (IGF) method [18], whereby ϕ is given by the convolution of ρ with the IGF [18]:

$$\phi_{ijk} = (\rho * G)_{ijk} = \sum_{i'j'k'} \rho_{i'j'k'} G(\mathbf{x}_{ijk} - \mathbf{x}_{i'j'k'}) \quad (3)$$

The IGF $G(\mathbf{x})$ is the integral of the Green function for Eq. (2), $\phi(x, y, z) = \gamma/4\pi\epsilon_0 \sqrt{x^2 + y^2 + \gamma^2 z^2}$, over one cell. It can be evaluated by taking sums and differences of a somewhat complicated

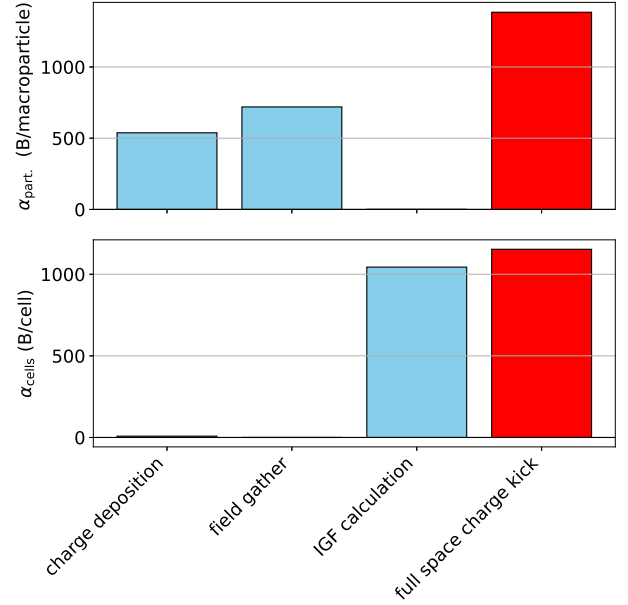


Figure 3: Coefficients for the increase in memory usage during the forward pass, as given by Eq. (1), when applied to a full space charge kick (red bars) or to selected parts of the space charge kick (blue bars).

analytical function $g(x, y, z)$ [19]:

$$G(\mathbf{x}) = g\left(x + \frac{\Delta x}{2}, y + \frac{\Delta y}{2}, \gamma z + \frac{\gamma \Delta z}{2}\right) - g\left(x - \frac{\Delta x}{2}, y + \frac{\Delta y}{2}, \gamma z + \frac{\gamma \Delta z}{2}\right) - g\left(x + \frac{\Delta x}{2}, y - \frac{\Delta y}{2}, \gamma z + \frac{\gamma \Delta z}{2}\right) - g\left(x - \frac{\Delta x}{2}, y - \frac{\Delta y}{2}, \gamma z + \frac{\gamma \Delta z}{2}\right) + g\left(x + \frac{\Delta x}{2}, y - \frac{\Delta y}{2}, \gamma z - \frac{\gamma \Delta z}{2}\right) + g\left(x - \frac{\Delta x}{2}, y + \frac{\Delta y}{2}, \gamma z - \frac{\gamma \Delta z}{2}\right) + g\left(x - \frac{\Delta x}{2}, y - \frac{\Delta y}{2}, \gamma z + \frac{\gamma \Delta z}{2}\right) - g\left(x - \frac{\Delta x}{2}, y - \frac{\Delta y}{2}, \gamma z - \frac{\gamma \Delta z}{2}\right) \quad (4)$$

with

$$g(x, y, z) = \frac{1}{4\pi\epsilon_0} \left[yz \operatorname{arcsinh}\left(\frac{x}{\sqrt{y^2 + z^2}}\right) + xz \operatorname{arcsinh}\left(\frac{y}{\sqrt{x^2 + z^2}}\right) + xy \operatorname{arcsinh}\left(\frac{z}{\sqrt{x^2 + y^2}}\right) - \frac{z^2}{2} \arctan\left(\frac{xy}{x\sqrt{x^2 + y^2 + z^2}}\right) - \frac{y^2}{2} \arctan\left(\frac{xz}{y\sqrt{x^2 + y^2 + z^2}}\right) - \frac{x^2}{2} \arctan\left(\frac{yz}{x\sqrt{x^2 + y^2 + z^2}}\right) \right] \quad (5)$$

A direct implementation of the convolution in Eq. (3) would involve $O(N_{\text{cells}}^2)$ operations. Instead, Cheetah (like many other beam tracking codes) uses the Hockney method [8] to evaluate Eq. (3). This method leverages the fact that convolutions can be evaluated efficiently using FFTs, and consists in the following steps:

- Compute the IGF $G(x)$ on the spatial grid using Eqs. (4) and (5)
- Compute the Fourier transform of G and ρ , using the FFT algorithm: $\hat{G} = \text{FFT}(G)$, $\hat{\rho} = \text{FFT}(\rho)$.
- Perform the element-wise multiplication of $\hat{\rho}$ and $\hat{G}$ in Fourier space: $\hat{\phi} = \hat{\rho}\hat{G}$
- Obtain ϕ by computing the inverse Fourier transform of $\hat{\phi}$: $\phi = \text{FFT}^{-1}(\hat{\phi})$

The first step in the above (computing the IGF $G(x)$ on the spatial grid) consists in a series of *element-wise* operations (given by Eq. (4) and Eq. (5)) on arrays of size N_{cells} . It is thus expected that the size of the data recorded during the forward pass scales proportionally to N_{cells} – and that it does not depend on $N_{\text{part.}}$, since the macroparticles are not involved in these operations. This is indeed confirmed in Fig. 4. Note from Fig. 3 that the coefficient α_{cells} for the IGF calculation (rightmost blue bar in the bottom panel) accounts for most of the memory usage per cell in the full space charge kick (red bar in the bottom panel). This can be explained by the large amount of individual operations involved in evaluations of Eq. (4) and Eq. (5), and the fact that, during the forward pass, some data needs to be recorded for each of these individual operations.

Similarly, the third step in the Hockney method (multiplication of $\hat{\rho}$ and $\hat{G}$ in Fourier space) is also an element-wise operation on arrays of size $O(N_{\text{cells}})$, and thus the amount of data recorded during the forward pass is again expected to scale proportionally to N_{cells} . On the other hand, the second and fourth steps involve FFTs which are *not* element-wise operations.

When measuring the change in memory usage during these FFTs and inverse FFT (using the tools described in App. C), we found that *no additional data* is recorded during these parts of the forward pass – in other words $\Delta M = 0$ irrespective of N_{cells} . This implies that no additional data is needed in order to propagate the gradients through an FFT, during the backward pass. Indeed a Fourier transform can be written as:

$$\hat{F}_{\ell'} = \sum_{\ell=1}^N e^{-2i\pi \frac{\ell\ell'}{N}} F_{\ell}$$

and applying the chain rule to propagate the gradients during the backward pass yields:

$$\frac{\partial f}{\partial F_{\ell}} = \sum_{\ell'=1}^N \frac{\partial f}{\partial \hat{F}_{\ell'}} \frac{\partial \hat{F}_{\ell'}}{\partial F_{\ell}} = \sum_{\ell'=1}^N \frac{\partial f}{\partial \hat{F}_{\ell'}} e^{-2i\pi \frac{\ell\ell'}{N}}$$

Hence the operation that propagates the gradients during the backward pass is itself a Fourier transform. (This can be confirmed for instance by inspecting the code of open-source packages that implement reverse-mode differentiation for FFTs [26].) This operation does not require additional input data beyond the gradients to be propagated $\partial f / \partial \hat{F}_{\ell'}$ (compare this with some of the operations in App. A that *do* require additional input data to propagate gradients in the backward pass), and thus this explains why no additional data is recorded when performing FFTs during the forward pass.

Overall, this section showed that the field solver is a combination of element-wise operation (which cause memory usage to increase proportionally to N_{cells} during the forward pass) and of Fourier transforms (which cause no increase in memory usage during the forward pass). Taken together, these observations indicate that the

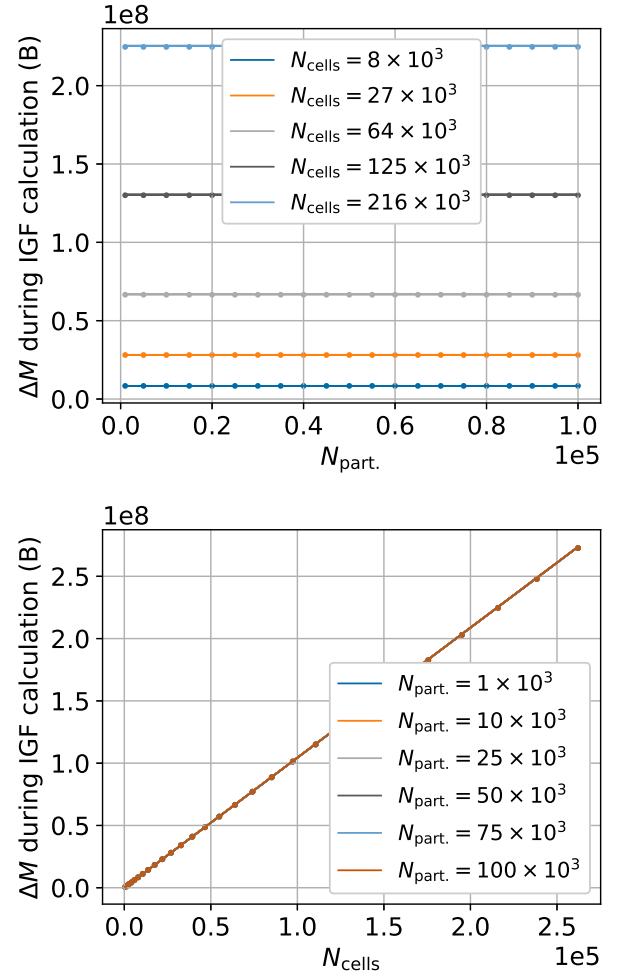


Figure 4: Increase in memory usage ΔM incurred during the calculation of the Integrated Green Function (IGF), as part of one space charge kick (forward pass, with reverse-mode differentiation turned on), as a function of the number of macroparticles and cells. Eq. (1) is fitted to the data, and the corresponding coefficients are represented in Fig. 3. The lines show the evaluation of this fitted formula for fixed number of cells (top) and fixed number of macroparticles (bottom).

increase in memory usage for the field solver, during the forward pass, scales like $O(N_{\text{cells}})$.

It is worth noting that, beyond FFT-based solvers, real-space iterative solvers (e.g. multigrid solvers, GMRES, as well as multigrid-preconditioned Krylov solvers) are also commonly used in beam tracking codes – due to their capacity to handle beam pipes with complex geometries. While the FFT operation does not incur *any* memory increase during the forward pass, iterative solvers likely do not possess a similar property. Consequently, their memory usage scaling is expected to deviate from the linear $O(N_{\text{cells}})$ behavior observed for FFT-based solvers. However, a comprehensive analysis

of the memory scaling of iterative solvers falls outside the scope of this work.

3.4 Memory scaling of the charge deposition

Fig. 5 shows the increase in memory usage measured during charge deposition (using the profiling tools described in App. C) as part of the forward pass. As can be observed, the memory usage depends strongly on the number of macroparticles ($\alpha_{\text{part.}} \approx 550$ B per macroparticle) and increases only mildly as a function of the number of cells ($\alpha_{\text{cells}} \approx 8$ B per cell). We observe no evidence for a dependency of the form $N_{\text{part.}} \times N_{\text{cells}}$, despite the fact that the formula

$$\rho_{ijk} = \frac{1}{\Delta x \Delta y \Delta z} \sum_{p=1}^{N_{\text{part.}}} q_p S(\mathbf{x}_{ijk} - \mathbf{x}_p)$$

involves both cells and particles. The reason is that this is a sparse operation, whereby $S(\mathbf{x}_{ijk} - \mathbf{x}_p)$ is a compact function which evaluates to zero for most grid points. When using Cloud-In-Cell functions for S (i.e., first-order splines), each macroparticle indeed only contributes to its two nearest grid points along each dimension (i.e., it contributes to 8 grid points in 3D). In practice, in Cheetah, this is implemented with the following steps:

- Compute the weighting coefficients

$$S(\mathbf{x}_{ijk} - \mathbf{x}_p) \equiv \left(1 - \frac{|\mathbf{x}_i - \mathbf{x}_p|}{\Delta x}\right) \times \left(1 - \frac{|\mathbf{y}_j - \mathbf{y}_p|}{\Delta y}\right) \times \left(1 - \frac{|\mathbf{z}_k - \mathbf{z}_p|}{\Delta z}\right)$$

for the 8 grid points surrounding each macroparticle. This results in a set of $8 \times N_{\text{part.}}$ coefficients.

- Add these contributions at the proper indices in the array ρ , using the PyTorch function `index_put_`.

Since the above steps generally involve $O(N_{\text{part.}})$ operations, it seems natural that the data recorded during the forward pass scales like $O(N_{\text{part.}})$. Further investigation – not shown here – revealed that the weak dependency with N_{cells} comes from data recorded during auxiliary operations on the array ρ (allocation, normalization by $1/\Delta x \Delta y \Delta z$).

3.5 Memory scaling of the field gather

The field gather operation is structurally similar to the charge deposition and consists in the following steps:

- Compute the weighting coefficients $S(\mathbf{x}_{ijk} - \mathbf{x}_p)$ for the 8 grid points surrounding each macroparticle.
- Sum the contributions to $F(\mathbf{x}_p)$ from the 8 nearest grid point for each macroparticle, using the PyTorch function `scatter_add`.

The increase in memory usage during field gather is shown in Fig. 6. Again, we observe a strong dependency with $N_{\text{part.}}$ (and in this case no dependency on N_{cells}). As can be seen in Fig. 3, charge deposition and field gather account for most of the memory consumption per macroparticle, during the forward pass of a full space charge kick.

4 Conclusion

In this paper, we studied the scaling of the increase in memory usage, during the forward pass of a beam tracking simulation using space charge with reverse-mode differentiation. Overall, we found

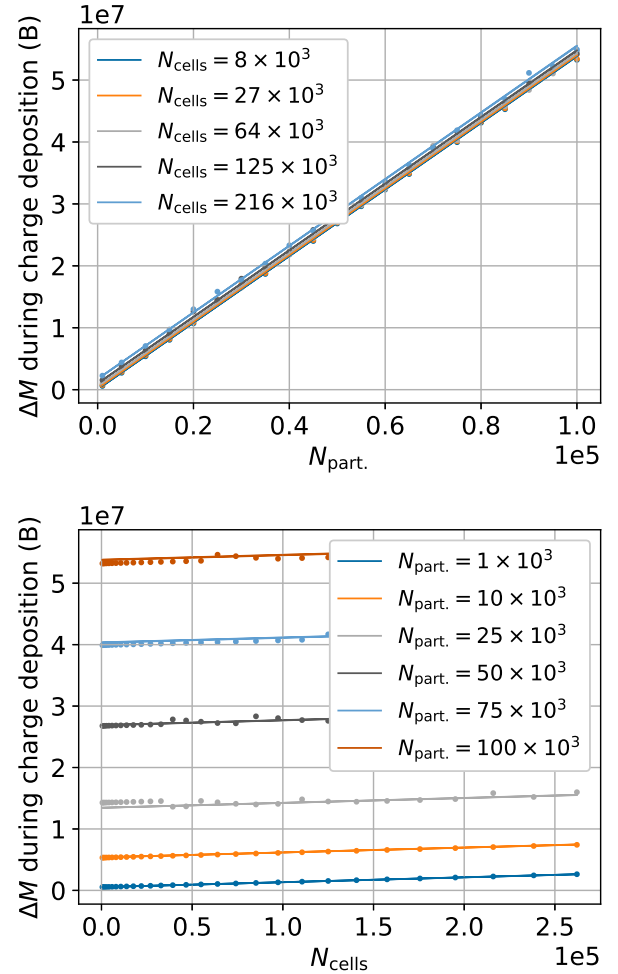


Figure 5: Increase in memory usage ΔM incurred during charge deposition, as part of one space charge kick (forward pass, with reverse-mode differentiation turned on), as a function of the number of macroparticles and cells. Eq. (1) is fitted to the data, and the corresponding coefficients are represented in Fig. 3. The lines show the evaluation of this fitted formula for fixed number of cells (top) and fixed number of macroparticles (bottom).

that the memory scaling is linear in the number of macroparticles $N_{\text{part.}}$ and cells N_{cells} , and that it increases with the number of space charge kicks applied to the beam throughout the simulation N_{kicks} :

$$\Delta M \simeq (\alpha_{\text{part.}} N_{\text{part.}} + \alpha_{\text{cells}} N_{\text{cells}}) \times N_{\text{kicks}} \quad (6)$$

The fact that the memory usage grows linearly in $N_{\text{part.}}$ and N_{cells} (and not, e.g., quadratically) suggests that the memory cost of space charge with reverse-mode differentiation is not prohibitively expensive – provided that the number of discrete space charge kicks N_{kicks} is not too large. For applications that require a large number of space charge kicks (such as beam tracking in a ring over

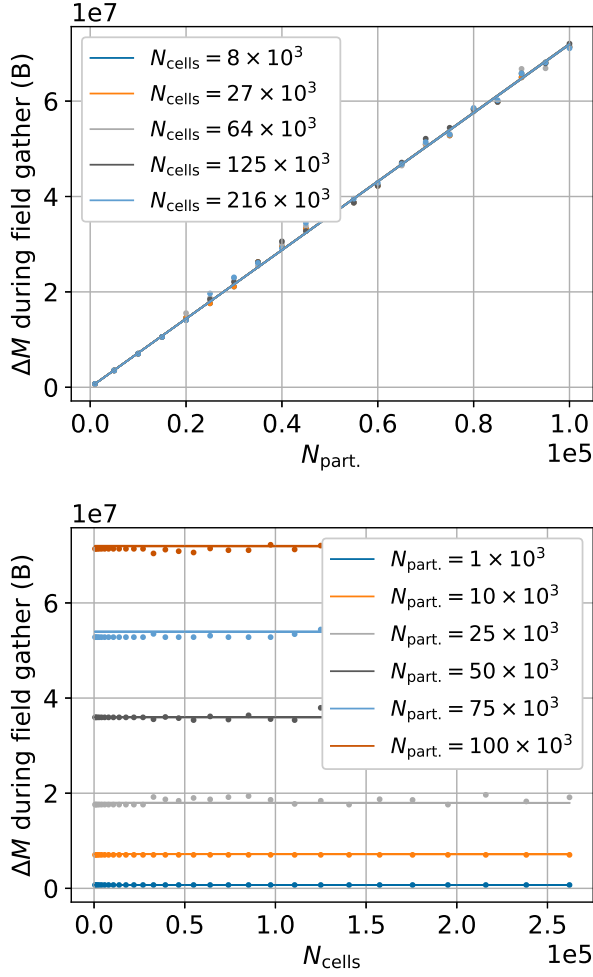


Figure 6: Increase in memory usage ΔM incurred during field gather, as part of one space charge kick (forward pass, with reverse-mode differentiation turned on), as a function of the number of macroparticles and cells. Eq. (1) is fitted to the data, and the corresponding coefficients are represented in Fig. 3. The lines show the evaluation of this fitted formula for fixed number of cells (top) and fixed number of macroparticles (bottom).

thousands of turns or more), the issues of growing memory usage can potentially be mitigated by using *gradient checkpointing* [14], a technique that reduces memory consumption at the expense of additional computation – by storing only a limited subset of data during the forward pass (in special checkpoints) and recomputing portions of the forward pass from these checkpoints during the backward pass.

The PIC methods employed here (namely charge deposition and field gather with compact shape factors, and FFT-based convolution for the field solve) are standard components shared by many beam tracking codes. As such, we expect the finding of linear memory

scaling with simulation resolution to generalize beyond Cheetah to any code implementing FFT-based space charge with reverse-mode differentiation. This expectation extends naturally to other collective effects, such as wakefields or coherent synchrotron radiation, which are typically handled with analogous numerical techniques. In this sense, the linear scaling law reported here should be understood as a general property of reverse-mode differentiation applied to FFT-based collective effect solvers, rather than a result specific to Cheetah.

Nevertheless, the numerical values of the memory coefficients are, of course, implementation-dependent. In the specific case of Cheetah, we find that the memory overhead per space charge kick (see Fig. 3) amounts to $\alpha_{\text{part.}} = 1.4$ kB per macroparticle – dominated by data recorded during charge deposition and field gather – and $\alpha_{\text{cells}} = 1.2$ kB per cell, dominated by the evaluation of the IGF. While these specific values apply to the current version of Cheetah and will differ across codes and implementations, they serve to illustrate how the linear scaling law can be used in practice to predict memory consumption and assess feasibility before running a simulation. Furthermore, regardless of the specific implementation, this work suggests that future memory optimization efforts, in Cheetah or in other codes, should prioritize the evaluation of the IGF analytical expression, as well as the charge deposition and field gather operations, since these consistently appear to dominate the memory cost of reverse-mode differentiation.

Acknowledgments

This work was supported by the Laboratory Directed Research and Development Program of Lawrence Berkeley National Laboratory, as well as by the U.S. Department of Energy, Office of Science, Office of High Energy Physics, General Accelerator R&D (GARD), under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was partially funded in the context of the 205 R&D program of the European XFEL.

5 Data availability statement

The scripts and instructions needed to reproduce these results are available on Zenodo.org at <https://doi.org/10.5281/zenodo.17822325>.

This research used the following software versions:

- CUDA 12.6
- CPython 3.13.5
- PyTorch 2.7.1
- A modified version of Cheetah v0.7.4, which was instrumented for memory profiling (see the Zenodo archive)

A Illustration of the memory requirements of reverse-mode differentiation, in a simple example

In order to illustrate why memory usage increases during the forward pass of reverse-mode differentiation, we consider a simple example:

$$f(\mathbf{x}) = ||\log(M\mathbf{x})||^2$$

where $\mathbf{x}$ is a vector of size N , M is a matrix of size $N \times N$, and where we wish to compute the derivative of f with respect to $\mathbf{x}$, $\partial f / \partial x_i$ for $i \in [1, N]$.

In this case, the forward pass proceeds in three steps:

- Step 1: compute $\mathbf{a} = M\mathbf{x}$ (matrix multiplication), and keep M in memory ($N \times N$ scalars)
- Step 2: compute $\mathbf{b} = \log(\mathbf{a})$ (element-wise operation), and keep the $1/a_i$ in memory (N scalars)
- Step 3: compute $f = \|\mathbf{b}\|^2 \equiv \sum_{i=1}^N b_i^2$, and keep $\mathbf{b}$ in memory (N scalars)

The reason why different variables needed to be kept in memory becomes apparent when considering the backward pass. In the backward pass, the previous steps are revisited in reverse order, to calculate the derivatives of f with respect to $\mathbf{b}$, $\mathbf{a}$ and finally $\mathbf{x}$ using the chain rule:

- From step 3: $\frac{\partial f}{\partial b_i} = 2b_i$, which is known because $\mathbf{b}$ was kept in memory. After computing $\frac{\partial f}{\partial b_i}$, the memory that held $\mathbf{b}$ can be deallocated.
- From the chain rule applied to step 2: $\frac{\partial f}{\partial a_i} = \frac{\partial f}{\partial b_i} \frac{\partial b_i}{\partial a_i}$. $\frac{\partial f}{\partial b_i}$ is known from the previous step of the backward pass, and by property of the log, $\frac{\partial b_i}{\partial a_i} = \frac{1}{a_i}$, which is known since the $1/a_i$ were kept in memory. After computing $\frac{\partial f}{\partial a_i}$, the memory that held the $1/a_i$ can be deallocated.
- From the chain rule applied to step 1: $\frac{\partial f}{\partial x_i} = \sum_{j=1}^N \frac{\partial f}{\partial a_j} \frac{\partial a_j}{\partial x_i}$. $\frac{\partial f}{\partial a_j}$ is known from the previous step of the backward pass, and by property of the matrix multiplication, $\frac{\partial a_j}{\partial x_i} = M_{ji}$, which is known because the matrix M was kept in memory. After computing $\frac{\partial f}{\partial x_i}$, the memory that held M can be deallocated.

Of course the above is a very simple example, and a full simulation consists of a much longer chain of operations. But conceptually each of them can be thought of as going from a set of intermediate variables $\{a_i\}$ to a new set of intermediate variables $\{a'_j\}$ (e.g., matrix multiplication, Fourier transform, element-wise addition/multiplication over an array, etc.). Then in order for reverse-mode differentiation to work, the code needs to define how to go from the $\partial f / \partial a'_j$ to the $\partial f / \partial a_i$ during the backward pass, and it needs to store the corresponding necessary variables during the forward pass (for instance, in PyTorch this can be done for custom operations by defining the methods `forward` and `backward` of a class derived from `torch.autograd.Function`, and by using the function `save_for_backward` in the `forward` method). In practice, different operations may require storing different amounts of memory in the forward pass, as seen in the above example (N^2 scalars for step 1, but N scalars for step 2).

B Benchmarks for the accuracy of Cheetah's space-charge implementation

B.1 Space charge fields in a Gaussian bunch

In order to benchmark the space charge solver in Cheetah, we consider the calculation of the Lorentz force due to space charge within a relativistic Gaussian beam. For this benchmark, the beam

has a charge $Q = 1$ nC, a transverse RMS size $\sigma_x = \sigma_y = 1$ mm, and a longitudinal RMS size $\sigma_z = 1$ μ m. We vary the beam's Lorentz factor γ from 10 to 10,000, and thus the aspect ratio of the beam in its rest frame $r = \sigma'_z / \sigma'_x = \gamma \sigma_z / \sigma_x$ varies from 0.01 to 10. This benchmark is therefore similar to the one performed in [12, 13].

In the beam's rest frame, the potential is given by [12]:

$$\phi'(x', y', z') = \frac{Q}{4\pi\epsilon_0} \sqrt{\frac{2}{\pi}} \int_0^\infty \frac{e^{-\frac{\lambda^2 x'^2}{2(\lambda^2 \sigma_x'^2 + 1)} - \frac{\lambda^2 y'^2}{2(\lambda^2 \sigma_y'^2 + 1)} - \frac{\lambda^2 z'^2}{2(\lambda^2 \sigma_z'^2 + 1)}}}{\sqrt{(\lambda^2 \sigma_x'^2 + 1)(\lambda^2 \sigma_y'^2 + 1)(\lambda^2 \sigma_z'^2 + 1)}} d\lambda$$

where primed quantities indicate values in the rest frame. Since $\mathbf{B} = \mathbf{0}$ for the space-charge field in the beam frame, the Lorentz force is $\mathbf{F}' = -q\nabla_{\mathbf{x}'}\phi'$. Using a Lorentz transform at fixed time $t = 0$ in the laboratory frame (for which $x', y', z' = x, y, \gamma z$), we can readily find the Lorentz force in the laboratory frame:

$$F_x(x, y, z) = \frac{F'_x(x', y', z')}{\gamma} = \frac{qQx}{4\pi\gamma\epsilon_0} \sqrt{\frac{2}{\pi}} \int_0^\infty \frac{\lambda^2 e^{-\frac{\lambda^2 x^2}{2(\lambda^2 \sigma_x^2 + 1)} - \frac{\lambda^2 y^2}{2(\lambda^2 \sigma_y^2 + 1)} - \frac{\lambda^2 \gamma^2 z^2}{2(\lambda^2 \gamma^2 \sigma_z^2 + 1)}}}{\sqrt{(\lambda^2 \sigma_x^2 + 1)^3 (\lambda^2 \sigma_y^2 + 1) (\lambda^2 \gamma^2 \sigma_z^2 + 1)}} d\lambda \quad (7)$$

$$F_z(x, y, z) = F'_z(x', y', z') = \frac{qQ\gamma z}{4\pi\epsilon_0} \sqrt{\frac{2}{\pi}} \int_0^\infty \frac{\lambda^2 e^{-\frac{\lambda^2 x^2}{2(\lambda^2 \sigma_x^2 + 1)} - \frac{\lambda^2 y^2}{2(\lambda^2 \sigma_y^2 + 1)} - \frac{\lambda^2 \gamma^2 z^2}{2(\lambda^2 \gamma^2 \sigma_z^2 + 1)}}}{\sqrt{(\lambda^2 \sigma_x^2 + 1)(\lambda^2 \sigma_y^2 + 1)(\lambda^2 \gamma^2 \sigma_z^2 + 1)^3}} d\lambda \quad (8)$$

These theoretical formulas are plotted in Fig. 7 (solid lines) and compared with the predictions of Cheetah's space charge solver (dots). The Cheetah calculation used 10^6 macroparticles and a grid extending over $\pm 6\sigma_x$, $\pm 6\sigma_y$, $\pm 6\sigma_z$ in x , y and z respectively, with 128^3 cells. As can be seen in Fig. 7, excellent agreement is found between the theoretical formulas and the Cheetah predictions.

B.2 Free expansion of a cold uniform bunch

We also evaluate the accuracy of Cheetah for the case of the free expansion of a cold, uniformly charged bunch. The setup is identical to that described in Sec.2 (see also [13]), and we use the same numerical and physical parameters listed in Table 1, with the exception of N_{slices} which is set to $N_{\text{slices}} = 10$ here for increased accuracy.

With these parameters, the bunch is spherical in its rest frame, and thus the space charge forces are linear inside the bunch ($E \propto \mathbf{r}$ in the rest frame). As a result, the bunch remains spherical and uniformly charged as it expands, and its radius R evolves according to the following envelope equation (with no external focusing and zero emittance) [21]:

$$\frac{d^2 R}{ds^2} = \frac{N_b r_c}{\gamma^2 \beta^2 R^2} \quad (9)$$

where R is the beam radius, N_b is the number of beam particles, r_c is the classical electron radius, and γ and β are the Lorentz factor and normalized velocity of the beam, respectively.

The top panel of Fig. 8 compares the results of the numerical integration of Eq. (9) with results obtained using Cheetah. In the case of Cheetah, the beam radius is determined using $R = \sqrt{5\langle x^2 \rangle}$,

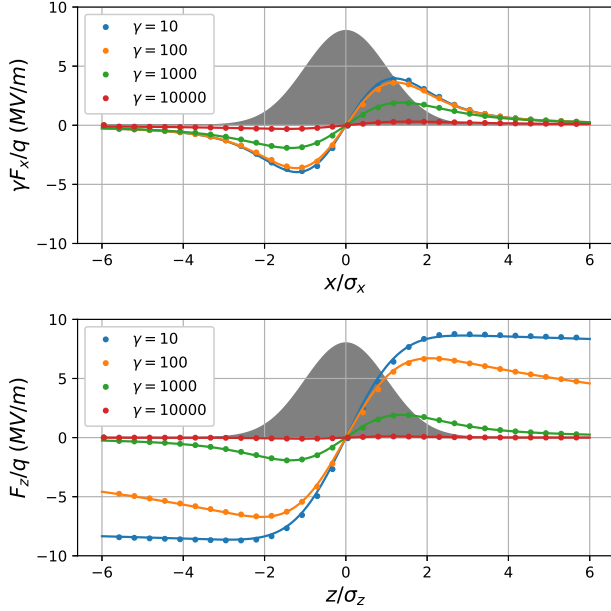


Figure 7: Plots of the Lorentz force $F = q(E + v \times B)$ felt by particles of a relativistic beam in the lab frame, due to space charge, for various values of the Lorentz factor γ . The solid lines are obtained by evaluating Eq. (7) and Eq. (8), while the dots are from the space charge calculation within Cheetah.

which is valid for a uniformly charged, spherical beam, where $\langle \dots \rangle$ denotes an average over macroparticles. In addition, in order to benchmark the accuracy of Cheetah’s automatic differentiation, we also compare dR/ds in the lower panel of Fig. 8. We emphasize that, in the case of Cheetah, the values of dR/ds on Fig. 8 are not obtained by an approximate numerical method (e.g., finite difference from the obtained values of $R(s)$), but instead the exact derivatives dR/ds are obtained using Cheetah’s built-in automatic differentiation. As can be seen in Fig. 8, there is excellent agreement between the predictions of the envelope equation and those of Cheetah, for both $R(s)$ (upper panel) and dR/ds (lower panel).

C Measuring memory usage

In this paper, the memory usage of Cheetah was measured using two built-in PyTorch tools for measuring memory use on GPUs. First, PyTorch’s `torch.profiler.profile` was used to generate the memory graphs shown in Fig. 1. This profiler includes the necessary traces to generate a memory graph and annotate it.

Second, `torch.cuda.memory_allocated()` was used to determine the memory before and after each selected part of the space charge kick to measure the ΔM plotted in Figs. 2 to 6. Care was taken to ensure that the memory was recorded before and after function calls and that Python’s `del` operator was used on temporary variables so that any vestigial memory allocations were freed by Python’s garbage collector and not included in the data.

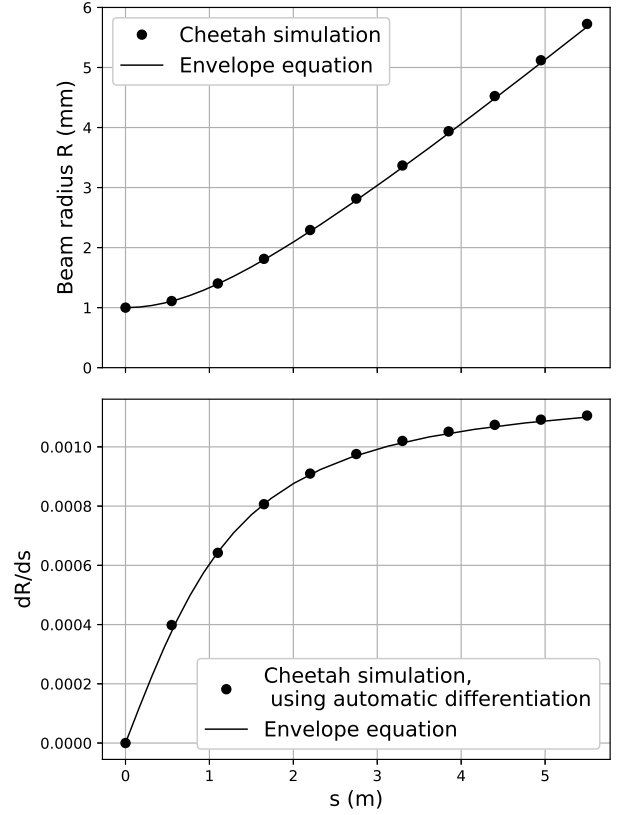


Figure 8: Comparison of beam radius evolution during the free expansion of a cold, uniformly charged bunch, as predicted by numerical integration of the envelope equation Eq. (9) and by Cheetah simulations. The setup uses the parameters from Table 1 with $N_{\text{slices}} = 10$.

D Fitting technique

In order to generate the linear fits for the data in Figs. 2 to 6, we ran least squares. Ordinary least squares (OLS) minimizes the sum of the squared absolute errors of the fit at each point. Given A is the feature matrix corresponding to the data, y_i are the outputs, we compute the coefficient vector x_{ols} for the OLS fit as follows:

$$x_{\text{ols}} = \underset{x}{\operatorname{argmin}} \sum_i |Ax - y_i|^2$$

Because both the independent variables and the output vary over orders of magnitude, OLS would heavily favor the data points at the extremes of the data set where the output has a high magnitude. This would cause the fit to essentially ignore the data points at the smaller end of the data set. To generate a fit that works in both the extremes and the smaller end of the data set, we instead minimize the sum of the squared *relative* errors of the fit at each point. The coefficient vector for this weighted least squares fit, x_{wls} , can be

computed as follows:

$$x_{\text{wls}} = \operatorname{argmin}_x \sum_i \left| \frac{Ax - y_i}{y_i} \right|^2 = \operatorname{argmin}_x \sum_i \left| \frac{1}{y_i} Ax - 1 \right|^2$$

This can be seen as a OLS fit with each row of the feature matrix scaled by the inverse y_i and the output set to a vector of ones. This weighted least squares fit was used instead of OLS in this paper.

References

- [1] C. K. Birdsall and A. B. Langdon. 2004. *Plasma Physics via Computer Simulation*. CRC Press. doi:10.1201/9781315275048
- [2] Tobias Boltz, Jose L. Martinez, Connie Xu, Kathryn R. L. Baker, Zihan Zhu, Jenny Morgan, Ryan Roussel, Daniel Ratner, Brahim Mustapha, and Auralee L. Edelen. 2025. Leveraging prior mean models for faster Bayesian optimization of particle accelerators. *Scientific Reports* 15, 1 (10 April 2025), 12232. doi:10.1038/s41598-025-95297-z
- [3] Kyle Cranmer, Johann Brehmer, and Gilles Louppe. 2020. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences* 117, 48 (2020), 30055–30062. arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1912789117 doi:10.1073/pnas.1912789117
- [4] Tommaso Dorigo, Andrea Giammanco, Pietro Vischia, Max Ahle, Mateusz Bawaj, Alexey Boldyrev, Pablo de Castro Manzano, Denis Derkach, Julien Donini, Auralee Edelen, Federica Fanzago, Nicolas R. Gauger, Christian Glaser, Atılım G. Baydin, Lukas Heinrich, Ralf Keidel, Jan Kieseler, Claudius Krause, Maxime Lagrange, Max Lamparth, Lukas Layer, Gernot Maier, Federico Nardi, Helge E.S. Pettersen, Alberto Ramos, Fedor Ratnikov, Dieter Röhrich, Roberto Ruiz de Austri, Pablo Martínez Ruiz del Arbol, Oleg Savchenko, Nathan Simpson, Giles C. Strong, Angela Taliencio, Mia Tosi, Andrey Ustyuzhanin, and Haitham Zaraket. 2023. Toward the end-to-end optimization of particle physics instruments with differentiable programming. *Reviews in Physics* 10 (2023), 100085. doi:10.1016/j.revip.2023.100085
- [5] Jeff Eldred, Valeri Lebedev, Kiyomi Seiya, and Vladimir Shiltsev. 2021. Beam intensity effects in Fermilab Booster synchrotron. *Phys. Rev. Accel. Beams* 24 (Apr 2021), 044001. Issue 4. doi:10.1103/PhysRevAccelBeams.24.044001
- [6] Juan Pablo Gonzalez-Aguilera, Young-Kee Kim, Ryan Roussel, Auralee Edelen, and Christopher Mayes. 2023. Towards fully differentiable accelerator modeling. *JACoW IPAC2023* (2023), WEPA065. doi:10.18429/JACoW-IPAC2023-WEPA065
- [7] Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives* (second ed.). Society for Industrial and Applied Mathematics. arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898717761 doi:10.1137/1.9780898717761
- [8] R. W. Hockney and J. W. Eastwood. 1988. *Computer simulation using particles*. Bristol: Hilger, 1988.
- [9] Austin Hoover and Jonathan C. Wong. 2024. High-dimensional maximum-entropy phase space tomography using normalizing flows. *Phys. Rev. Res.* 6 (Aug 2024), 033163. Issue 3. doi:10.1103/PhysRevResearch.6.033163
- [10] Jan Kaiser, Chenran Xu, Annika Eichler, and Andrea Santamaria Garcia. 2024. Bridging the gap between machine learning and particle accelerator physics with high-speed, differentiable simulations. *Phys. Rev. Accel. Beams* 27 (May 2024), 054601. Issue 5. doi:10.1103/PhysRevAccelBeams.27.054601
- [11] N. Kuklev, M. Wallbank, N. Banerjee, J. Jarvis, and A. Romanov. 2025. End-to-end differentiable digital twin for the IOTA/FAST facility. (2025).
- [12] Christopher Mayes, Robert Ryne, and David Sagan. 2018. 3D Space Charge in Bmad. In *9th International Particle Accelerator Conference*. doi:10.18429/JACoW-IPAC2018-THPAK085
- [13] C.E. Mitchell, M. Garten, A. Huebl, R. Lehe, J. Qiang, R.T. Sandberg, and J.-L. Vay. 2024. ImpactX Modeling of Benchmark Tests for Space Charge Validation. In *Proc. 68th Adv. Beam Dyn. Workshop High-Intensity High-Brightness Hadron Beams (HB'23) (ICFA Advanced Beam Dynamics Workshop on High-Intensity and High-Brightness Hadron Beams, 68)*. JACoW Publishing, Geneva, Switzerland, 583–586. doi:10.18429/JACoW-HB2023-THBP44
- [14] PyTorch Contributors. 2024. torch.utils.checkpoint. https://docs.pytorch.org/docs/stable/checkpoint.html. Accessed: 2024.
- [15] Ji Qiang. 2018. Three-dimensional envelope instability in periodic focusing channels. *Phys. Rev. Accel. Beams* 21 (Mar 2018), 034201. Issue 3. doi:10.1103/PhysRevAccelBeams.21.034201
- [16] Ji Qiang. 2023. Differentiable self-consistent space-charge simulation for accelerator design. *Phys. Rev. Accel. Beams* 26 (Feb 2023), 024601. Issue 2. doi:10.1103/PhysRevAccelBeams.26.024601
- [17] Ji Qiang. 2025. Study of fully coupled 3D envelope instability using automatic differentiation. https://arxiv.org/abs/2512.02433. (2025).
- [18] Ji Qiang, Steve Lidia, Robert D. Ryne, and Cecile Limborg-Deprey. 2006. Three-dimensional quasistatic model for high brightness beam dynamics simulation. *Phys. Rev. ST Accel. Beams* 9 (Apr 2006), 044204. Issue 4. doi:10.1103/PhysRevSTAB.9.044204
- [19] Ji Qiang, Chad Mitchell, Remi Lehe, and Arianna Formenti. 2024. Implementation of the Integrated Green's Function Method for 3D Poisson's Equation in a Large Aspect Ratio Computational Domain. *Journal of Software Engineering and Applications* 17, 9 (2024), 740–749.
- [20] Ji Qiang, Jinyu Wan, Allen Qiang, and Yue Hao. 2025. Fast chaos indicator from auto-differentiation for dynamic aperture optimization. https://arxiv.org/abs/2510.25196. (2025).
- [21] M. Reiser. 2008. *Theory and Design of Charged Particle Beams*. John Wiley & Sons, Ltd.
- [22] R. Roussel, A. Edelen, C. Mayes, D. Ratner, J. P. Gonzalez-Aguilera, S. Kim, E. Wisniewski, and J. Power. 2023. Phase Space Reconstruction from Accelerator Beam Measurements Using Neural Networks and Differentiable Simulations. *Phys. Rev. Lett.* 130 (Apr 2023), 145001. Issue 14. doi:10.1103/PhysRevLett.130.145001
- [23] Ryan Roussel, Juan Pablo Gonzalez-Aguilera, Eric Wisniewski, Alexander Ody, Wanming Liu, John Power, Young-Kee Kim, and Auralee Edelen. 2024. Efficient six-dimensional phase space reconstructions from experimental measurements using generative machine learning. *Phys. Rev. Accel. Beams* 27 (Sep 2024), 094601. Issue 9. doi:10.1103/PhysRevAccelBeams.27.094601
- [24] J.-L. Vay. 2008. Simulation of beams or plasmas crossing at relativistic velocity. *Physics of Plasmas* 15, 5 (02 2008), 056701. doi:10.1063/1.2837054
- [25] J. Wan, H. Alamprese, C. Ratcliff, J. Qiang, and Y. Hao. 2025. JuTrack: a Julia package for auto-differentiable accelerator modeling and particle tracking. *Comp. Phys. Comm.* 309 (2025), 109497.
- [26] Eric Wong. 2017. pytorch_fft: A PyTorch wrapper for CUDA FFTs. https://github.com/locuslab/pytorch_fft. Apache-2.0 License.
- [27] Takaaki Yasui, Susumu Igarashi, Yoichi Sato, Tadashi Koseki, and Kazuhito Ohmi. 2020. Transverse emittance growth caused by space-charge-induced resonance. *Phys. Rev. Accel. Beams* 23 (Jun 2020), 061001. Issue 6. doi:10.1103/PhysRevAccelBeams.23.061001
- [28] T. Zolkin, A. Burov, and B. Pandey. 2018. Transverse mode-coupling instability and space charge. *Phys. Rev. Accel. Beams* 21 (Oct 2018), 104201. Issue 10. doi:10.1103/PhysRevAccelBeams.21.104201