Differentiable simulations for particle tracking in accelerators: Analysis, benchmarking, and optimization

Francisco Huhn[*] and Francesco M. Velotti[†]
CERN, Geneva, Switzerland

(Received 14 July 2025; accepted 8 October 2025; published 25 November 2025)

Optimization of beamlines and lattices is a common problem in accelerator physics, which is usually solved with semianalytical methods and numerical optimization routines. However, these are usually of the gradient-free or finite-difference type, whose computational cost grows quickly with the number of optimization parameters. On the other hand, the cost of gradient-based optimization can scale well with the number of parameters, but only if the computation of the gradient is itself efficient (e.g., not via finite differences, which are inefficient). Recently, there has been an emergence of so-called "differentiable" codes that efficiently provide gradients. Nevertheless, analysis and benchmarking comparisons of these techniques have largely been absent from the literature. In this work, we develop our own differentiable code via autodifferentiation. We analyze and benchmark differentiability against finite differences in two test cases, a space-charge FODO cell from the literature and a realistic future beamline at CERN. The analyses and benchmarking are done both theoretically and empirically. Finally, we embed the gradient provided by the differentiable code in gradient-based optimization routines and compare with gradient-free methods. This work offers the first such analysis and benchmarking and thus contributes toward the development of more efficient and performant particle accelerators.

DOI: 10.1103/cbds-lst1

I. INTRODUCTION

Optimization of beamlines or lattices is a common problem in accelerator physics, in which one or a handful of objectives is minimized/maximized, potentially subject to constraints, by varying a number of parameters, such as quadrupole strengths, etc. Common objectives are Twiss functions matching [1], beam size minimization (e.g., Ref. [2]), beam-distribution tailoring [3], etc. Numerical optimization algorithms, in contrast with manual or grid search, have been successfully applied to efficiently solve these problems (e.g., Refs. [2,4]). These numerical routines can be provided by an external package and paired with an accelerator-physics simulation software, or be provided by the simulation tool itself. MAD-X [5] is among the most widely used simulation packages in accelerator physics and provides both gradient-based, via finite differences, and gradient-free, like Simplex [6], optimization routines. More recent tools, such as Xsuite [7], provide comparable functionality for lattice tuning. Nevertheless, for tracking-based optimization or for the global design of a lattice (e.g., element count and placement), practitioners predominantly adopt gradient-free methods, as the finite-difference evaluation of derivatives incurs a prohibitive computational cost. However, gradient-free algorithms suffer from the "curse of dimensionality" [8], i.e., the computational cost quickly becomes prohibitive as the number of parameters increases, rendering their application unfeasible in mid- to high-dimensional design spaces.[1]

On the other hand, gradient-based algorithms, which make use of the derivative of the objective (and constraints) with respect to the design parameters, in conjunction with the adjoint technique for gradient computation, are computationally efficient—they have been extensively applied in various fields, from fluid mechanics [9,10], where simulations can range from thousands to hundreds of millions of degrees of freedom, to machine learning [11], where models with up to billions of parameters are optimized. The use of the adjoint technique is critical, as naive computation of the gradient, such as via finite differences, is inefficient, because it suffers from the same curse of dimensionality as gradient-free methods (see Sec. II A). In this technique, the adjoint system is obtained and used to propagate adjoint vectors backward in time, which allows for an efficient computation

[*]Contact author: francisco.huhn@cern.ch
[†]Contact author: francesco.maria.velotti@cern.ch

Published by the American Physical Society under the terms of the Creative Commons Attribution 4.0 International license. Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

[1]What is considered mid- and high-dimensional depends on the problem and the computational resources available.

of the gradient (see Sec. II B). Traditionally, the adjoint system would be derived analytically and then numerically implemented in code. However, this approach has several disadvantages, being: (i) laborious, as deriving the adjoint system analytically can quickly become unwieldy; (ii) error prone, as the mathematical expressions must be manually transposed; and (iii) cumbersome to maintain, especially in updates (e.g., when a new feature is implemented). Alternatively, one can make use of autodifferentiation (see Sec. II C), which breaks down all computational operations into a graph of elementary operations, over which applying the chain rule becomes straightforward. This approach suffers from none of the aforementioned issues, but comes at the cost of building the graph, i.e., computational overhead, and storing and accessing it, i.e., memory cost. It is this method that has enabled and powered the successful training of large models in the field of machine learning and is provided by the most popular libraries, such as PyTorch [12], TensorFlow [13], JAX [14], etc.

In accelerator physics, the application of the adjoint technique (be it explicit or via autodifferentiation) is scarce. While new so-called "differentiable" simulation codes have emerged [15–18], they are still far from mainstream in the community. More importantly, use-case examples are often of the tutorial class, i.e., too simple, and comprehensive benchmarking and comparisons of the methods—both qualitative and quantitative—are largely absent from the literature.[2] In fact, the only empirical comparison between finite differences and autodifferentiation is found in a passage in [17], which states that autodifferentiation takes twice as long as (central) finite differences to compute the same gradient. However, a relevant detail absent from the paper, but mentioned in the code repository [19] is that only forward autodifferentiation was implemented. Since the problem has seven parameters and one objective, reverse (i.e., adjoint), not forward, autodifferentiation is likely the optimal method (see Sec. II B for a discussion on forward vs adjoint). Therefore, to the best of our knowledge, this publication represents the first time-performance benchmarking and comparison between finite differences and autodifferentiation, as well as between gradient-free and gradient-based optimization.

II. GRADIENT COMPUTATION

Before delving into the details of the computation of the gradient, let us define the problem. The beamline is composed of N elements. Starting from the initial state $\boldsymbol{x}_0 \in \mathbb{R}^{N_p \times 6}$ (if using six dimensions for each particle), where N_p is the number of particles, we can track the particles by iterating over the elements, i.e., by sequentially applying the (potentially nonlinear) transfer maps of each element

$$\boldsymbol{x}_n = \boldsymbol{f}_n(\boldsymbol{x}_{n-1}; \boldsymbol{p}_n), \tag{1}$$

where $\boldsymbol{f}_n$ is the transfer map and $\boldsymbol{p}_n$ the vector of parameters of the nth element. Equation (1) is called the primal and is what is commonly meant by "simulation."

We wish to minimize the objective

$$L(\boldsymbol{x}_0, \cdots, \boldsymbol{x}_N; \boldsymbol{p}_1, \cdots, \boldsymbol{p}_N), \tag{2}$$

which depends on the states, $\boldsymbol{x}_n$, and on the parameters, potentially explicitly (e.g., one might wish to add a penalty proportional to magnet strengths), but always implicitly via the states, as $\boldsymbol{x}_n = \boldsymbol{g}_n(\boldsymbol{x}_0; \boldsymbol{p}_1, \cdots, \boldsymbol{p}_n)$. For simplicity, but without loss of generality (generality can be achieved via composition),[3] we will assume going forward that only the first element has parameters, $\boldsymbol{p}_1 = \boldsymbol{p} = (p_1 \cdots p_m)$, and the objective function only depends on the final state of all the states, i.e., we have $L(\boldsymbol{x}_N; \boldsymbol{p})$.

A. Finite differences

In finite differences (FD), the gradient is computed by computing each partial derivative via a perturbation of the primal. For example, in the simplest case, forward FD, which approximates derivatives to first order:

$$\frac{\partial L}{\partial p_i} \approx \frac{L(\boldsymbol{x}_N; p_1, \cdots, p_i + \epsilon, \cdots, p_m) - L(\boldsymbol{x}_N, p_1, \cdots, p_m)}{\epsilon}. \tag{3}$$

To obtain the whole gradient, i.e., derivatives for all parameters, one must compute m perturbed primals and one unperturbed. In central FD (second order), $2m$ perturbed primals are needed. Generally, denoting the cost of one primal by C_P, the cost of FD is

$$C_\mathrm{FD} = (am + b) C_\mathrm{P}, \tag{4}$$

where the constants a and b depend on the type of FD (e.g., $a = 1, b = 1$ for forward, $a = 2, b = 0$ for central), i.e., the cost of computing the gradient is proportional to the number of parameters and cost of a primal.

B. Differentiation

Alternatively, let us attempt to compute the gradient via calculus

$$\frac{dL}{d\boldsymbol{p}} = \frac{\partial L}{\partial \boldsymbol{p}} + \frac{\partial L}{\partial \boldsymbol{x}_N} \frac{\partial \boldsymbol{x}_N}{\partial \boldsymbol{p}}. \tag{5}$$

[2]That adjoint has proved to be highly effective in other fields, it should not be assumed that it is also in accelerator physics, specifically in particle tracking (see Appendix B for a discussion).

[3]Computing gradients for an intermediate element reuses calculations for the first element. For the kth element, Eq. (6) is the same, stopping at $\partial f_k / \partial p | x_{k-1}$, with all $\partial f_n / \partial x | x_{n-1}$ terms to the left the same. In FD, instead of one extra term $(\partial f_k / \partial p | x_{k-1})$, a whole new primal must be evaluated from the kth element onward. Thus, the analysis for any set of elements mirrors that of the first.

The first term of the rhs is straightforward to compute. Focusing on the second term, we successively apply the chain rule, obtaining

$$\frac{\partial L}{\partial \boldsymbol{x}_N}\frac{\partial \boldsymbol{x}_N}{\partial \boldsymbol{p}} = \frac{\partial L}{\partial \boldsymbol{x}_N}\frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{x}}\bigg|_{\boldsymbol{x}_{N-1}} \cdots \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{p}}\bigg|_{\boldsymbol{x}_0}, \tag{6}$$

where the terms $\partial \boldsymbol{f}_n/\partial \boldsymbol{x}$ are the Jacobians of the transfer map of each element. Denoting the dimension of the system D (e.g., $D = 6N_p$), the first term of Eq. (6) is a $1 \times D$ vector. Each of the Jacobians is equivalent to a $D \times D$ matrix. Finally, the last term is a $D \times m$ matrix.

The rhs of Eq. (6) can be computed from right to left or left to right. Right to left is called forward[4] or tangent mode. In this case, we have an $N-1$-long series of $D \times D$ by $D \times m$ multiplications, followed by one $1 \times D$ by $D \times m$, resulting in a computational cost of

$$C_{\mathrm{C}} = ((N-1)D^2 m + Dm)c_{\mathrm{ma}}, \tag{7}$$

where c_{ma} is the cost of a scalar multiplication-addition. Left to right is called reverse or adjoint mode. Analogously, we have an $N-1$-long series of $1 \times D$ by $D \times D$ multiplications, followed by one $1 \times D$ by $D \times m$, resulting in a complexity of

$$C_{\mathrm{C}} = ((N-1)D^2 + Dm)c_{\mathrm{ma}}. \tag{8}$$

Clearly, from Eqs. (7) and (8), adjoint mode is advantageous as m is increased, since m is multiplied only by D, in contrast with $(N-1)D^2 + D$ in tangent. This is the case because we are considering the situation of one objective function and m parameters. If, instead, we had m objective functions and one parameter, the situation would be reversed. When the number of parameters is greater than the number of objectives, one should prefer the adjoint approach. Otherwise, the tangent should be preferred.

Irrespective of the approach, adjoint vs tangent, to calculate the gradient via Eq. (6), the states $\boldsymbol{x}_n$ are required (i.e., the primal must be computed), as well as the terms $\partial \boldsymbol{f}_n/\partial \boldsymbol{x}|_{\boldsymbol{x}_{n-1}}$, necessary to differentiate the transfer maps. Thus, the total cost of the method in terms of the cost of the primal is

$$\frac{C_{\mathrm{A}}}{C_{\mathrm{P}}} = \frac{C_{\mathrm{P}} + C_{\mathrm{D}} + C_{\mathrm{C}}}{C_{\mathrm{P}}}, \tag{9}$$

where C_{P}, C_{D}, and C_{C} are the costs of the primal, differentiation, and the computation of the chained derivative by multiplication-addition of the chain of the partial derivatives in adjoint mode. The cost of differentiation, C_{D}, is $O(C_{\mathrm{P}})$,[5] which we write, foregoing any dominated terms, as

[4]Not to be confused with forward FD, which is unrelated.

[5]For example, if the primal is $f(x,y) = xy$, then $\partial_p f = y\,\partial_p x + x\,\partial_p y$, i.e., $C_{\mathrm{D}} = 2C_{\mathrm{P}}$.

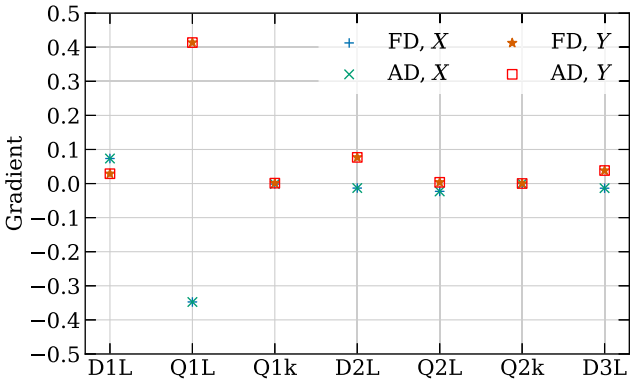


FIG. 1. Gradients of horizontal (X) and vertical (Y) emittances, via autodifferentiation (AD) and finite differences (FD).

$$C_{\mathrm{D}} = \kappa_{\mathrm{dp}} C_{\mathrm{P}}. \tag{10}$$

The relative cost of computing the chained derivative, $C_{\mathrm{C}}/C_{\mathrm{P}}$, depends on the problem, i.e., the primal, and the approach, tangent, or adjoint. For example, for a full linear primal, $C_{\mathrm{P}} = O(ND^2)$, whereas generally, nonlinearities will introduce higher computational complexity. Equation (8) for C_{C} and Eq. (10) can be substituted in Eq. (9) to obtain the total cost

$$\frac{C_{\mathrm{A}}}{C_{\mathrm{P}}} = 1 + \kappa_{\mathrm{dp}} + \kappa_{\mathrm{sp}} + \kappa_{\mathrm{mp}} m, \tag{11}$$

where $\kappa_{\mathrm{sp}} = c_{\mathrm{ma}}(N-1)D^2/C_{\mathrm{P}}$ and $\kappa_{\mathrm{mp}} = c_{\mathrm{ma}}D/C_{\mathrm{P}}$. Therefore, while strictly speaking, the adjoint's cost scales linearly with the number of parameters (similarly to FD), the linear coefficient, κ_{mp}, is small to negligible. Additionally, $\kappa_{\mathrm{sp}} \to 0$ as the computational complexity of the primal increases.[6] For example, even with a computationally simple, linear primal, $\kappa_{\mathrm{mp}} = c_{\mathrm{ma}}D/(c_{\mathrm{ma}}ND^2) = 1/(ND)$ and $\kappa_{\mathrm{sp}} = c_{\mathrm{ma}}(N-1)/(c_{\mathrm{ma}}N) = (N-1)/N$.

This highlights the big advantage of the adjoint method: computing the gradient on one parameter costs approximately the same as on any number of parameters[7] (κ_{mp} is very small); contrasting with FD, where the linear coefficient [a in Eq. (4)] is at least 1, in first-order FD, and larger in higher orders. For m objective functions and one parameter, analysis of the tangent approach, i.e., forward propagation, results in similar expressions.

C. Autodifferentiation

While one may derive the analytical expressions of $\partial \boldsymbol{f}_n/\partial \boldsymbol{x}_n$ and other terms, as mentioned in Sec. I, this

[6]How the function varies for given N (complexity), not as N increases (cost). For example, for order-n nonlinear primals, $C_{\mathrm{P}} = c_{\mathrm{ma}}ND^{n+1}$, and $\kappa_{\mathrm{sp}} = (N-1)/(ND^n) \to_{n\to\infty} 0$.

[7]Practical limitations, like memory, aside.

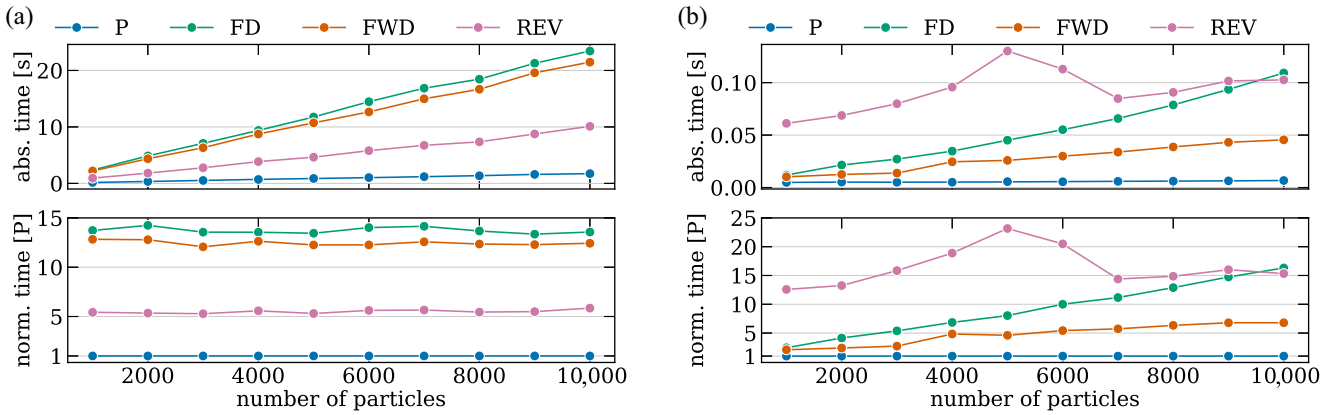


FIG. 2. Execution times of primal (P), finite differences (FD), adjoint autodifferentiation (REV), tangent autodifferentiation (FWD). Top panels: absolute time; bottom panels: normalized by primal time. (a) CPU and (b) GPU.

is laborious, error prone, and hard to maintain. Instead, autodifferentiation (AD) can be used. In AD, complex computations, e.g., the primal, are broken down into a directed acyclic graph of elementary operations, whose derivatives are known, over which the chain rule can be applied successively. An example of reverse AD (AD-REV) for a thin-sextupole kick is given in Appendix A1 and a cost analysis of the method is given in Appendix A2.

Autodifferentiation found great applicability and success in machine learning [20], with the most popular libraries (e.g., PyTorch [12], TensorFlow [13], JAX [14]) providing it as an out-of-the-box feature. In the majority of cases, very little setup is needed—one programs the primal as a function and then calls the autodifferentiation function (tangent or adjoint) on it. Moreover, these libraries provide GPU support, enabling the study of more computationally-demanding cases. While PyTorch has been the framework of choice for new differentiable codes [15,16], we implement ours [21] on top of JAX, which is designed for high-performance numerical computing and provides a NumPy-style API.

D. Computational time vs complexity

While it should be clear that above a certain number of parameters or primal computational complexity, autodifferentiation should be advantageous with respect to finite differences, it is not clear what this number is. Analysis of the expressions of the computational cost of each method would suggest it is small. However, the situation is complicated by parallelization. For example, the different perturbed primals of finite differences can be straightforwardly run in parallel. Parallelizing the adjoint is significantly harder. This is where the distinction between computational time and complexity becomes important. In the end, practitioners are interested in shorter computational times, not necessarily complexity. Thus, it may turn out that, despite having higher computational complexity, finite differences may be faster than the adjoint. Hence the necessity for the kind of benchmarking and comparison done in this work. It follows thus that the relationship between computation time and complexity is strongly dependent on the interplay between the problem (system size, density, etc.), the method/numerics (FD, AD, and their implementation) and the machine (e.g., CPU, GPU, memory, etc.). Our tests were run on a machine with Intel Xeon Silver 4110 CPU, 16 GB RAM, Tesla V100 GPU with 32 GB VRAM, which we believe is representative of those available to accelerator-physics practitioners, with all computations in double precision.

III. PROBLEMS AND RESULTS

A. FODO with space charge

1. Single cell

This case is taken from [17]. It is a 6D tracking problem of 5000 particles over a FODO cell with space-charge effects [22]. Following the reference, each of the drifts is sliced into segments of 0.05 m and the quadrupoles are sliced into segments of 0.025 m. The drift segments are integrated via drift-kick-drift, with the kick due to space-charge effects. The quadrupole segments are similarly halved, with space-charge kick applied between the halves, with each half being further subdivided into 20 subsegments, with each subsegment being integrated via the 7-step Yoshida integrator (fourth order). The objective functions are the horizontal and vertical emittances. There are seven parameters, five lengths (one per each of the three drifts and one per each of the two quadrupoles) plus two strengths (one per quadrupole).

We compare the values of the gradients obtained via AD (adjoint) and FD (central FD with step of 10^{-6} [17]). Figure 1 shows great agreement between the two methods and the values are also very similar to those found in [17].[8]

[8]The reason for the slight discrepancies is unknown.

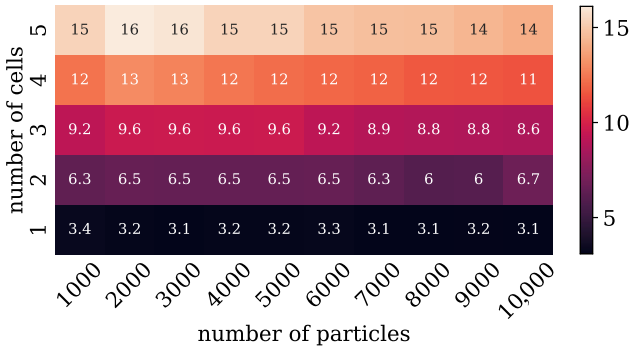


FIG. 3. Ratio of execution times: FD over AD-REV—CPU.

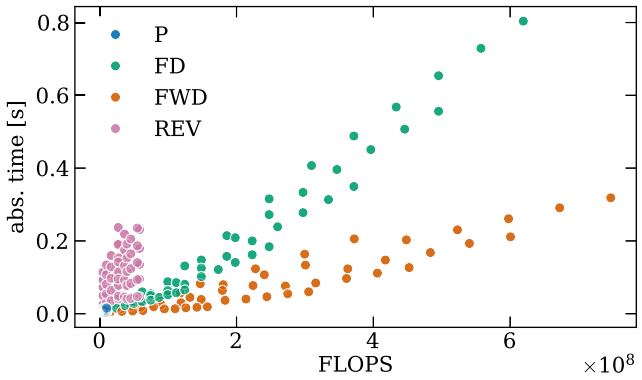


FIG. 5. Execution time vs number of operations.

Figure 2 shows the execution time for each method FD, AD-FWD, AD-REV, as well as the primal, on both CPU and GPU. On CPU, as the computational complexity (due to the number of particles) increases, the computational times increase. Normalizing by the cost of one primal (i.e., of one simulation), we can see that AD-REV costs the equivalent of 5 primals, while FD costs close to 3 times more. On the other hand, AD-FWD costs only slightly less than FD, which is not unsurprising given the low number of parameters. Notice, however, that this is in stark contrast with the findings in [17] that FD is twice as fast as AD-FWD. On GPU, however, there is a surprising reversal: AD-REV becomes the least-performing method for the majority of cases ($N_p \leq 9000$) and is only marginally faster than FD at $N_p = 10,000$. Nevertheless, autodifferentiation still comes ahead with AD-FWD. The reason for these, at first surprising, results is alluded to in Sec. II D—GPUs have great parallel processing power, which lends itself greatly to FD and AD-FWD, as both can easily take advantage of this capability due to their embarrassingly-parallel nature, and not as much to AD-REV (e.g., memory access costs). An analysis of time vs complexity in the following section (Fig. 5) effectively demonstrates this. This is an advantage that should subside with increasing problem complexity—indeed, at $N_p = 10,000$, we already observe a reversal in the relative performance of FD vs AD-REV.

2. *Multiple cells*

We slightly modify the FODO cell of the previous section. To slightly reduce the computational cost of each cell, we reduce the number of integration steps of a half slice from 20 to 5. To avoid two consecutive drifts, we double the length of the first drift and remove the last drift. Taking this modified cell, we make a beamline of up to 5 cells. Thus, we not only add computational complexity to the problem, similarly to adding more particles, but also increase the number of parameters. With this increase, the performance gap between FD and AD should become larger (see Sec. II B). The results for CPU (Fig. 3) show that, as in the previous section, AD-REV is 3 times faster than FD for 1 cell, but for 5 cells it is 15 times faster. This is not surprising, since the computational cost of AD-REV is practically constant in primal terms, whereas that of FD increases linearly with the number of parameters. On the other hand, if the code is executed on GPU [Fig. 4(a)], FD is faster than AD-REV for cases with a low number of cells and a low number of particles. Once again, this shows the potential of FD to take advantage of parallelization in "small" problems. However, in these cases, AD-FWD is even faster than FD [Fig. 4(b)]. The computational time vs cost, i.e., the number of operations, distinction can be clearly seen in Fig. 5, where time is plotted vs an estimate

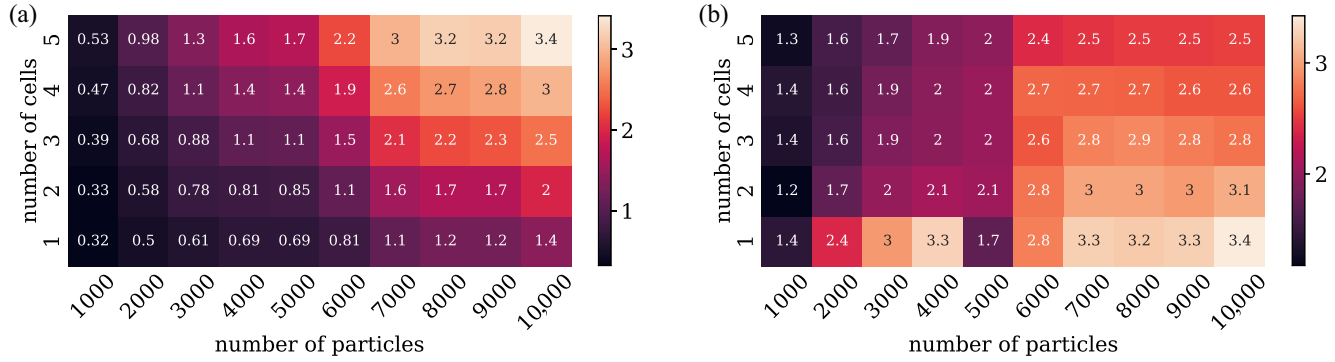


FIG. 4. Ratio of execution times: FD over AD—GPU. (a) AD-REV and (b) AD-FWD.

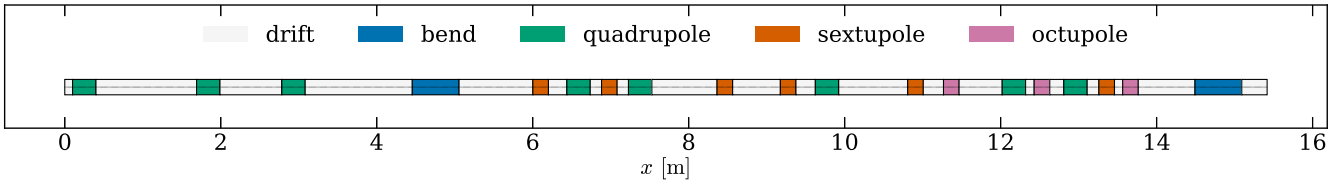


FIG. 6. AWAKE line.

TABLE I. Elements of the AWAKE line. NoE and NoP stand for number of elements and parameters.

Element type	Parameters	NoE	NoP
Drift	Length	20	20
Sector bend	Length, angle	2	4
Quadrupole	Length, strength	8	16
Sextupole	Length, strength	6	12
Octupole	Length, strength	3	6
Total		39	58

of the number of operations.[9] For the same number of operations, AD-REV takes longer than FD and AD-FWD.

In essence, for the cases analyzed, AD-REV is generally faster than FD, both on CPU and GPU. When solving small cases on GPU, AD-REV may be slower than FD, but AD-FWD is still faster than FD, which means that autodifferentiation is more efficient than finite differences in all cases tested.

B. Realistic beamline

To obtain a more realistic estimate of each method's performance, we have chosen a high-energy beamline so that collective effects can be safely neglected,[10] incorporating both sextupole and octupole magnets. This is the 150 MeV electron transfer line now being designed for the next phase of the CERN AWAKE experiment; it is designed to deliver micrometer-scale transverse beam sizes at the plasma injection point for subsequent acceleration [2].

Unlike the problem in Sec. III A, there are no collective effects. In this case, far more typical, each particle evolves independently—the joint system (i.e., system of all particles) is a block-diagonal system, each block corresponding to a particle. The beamline (Fig. 6) is composed of drifts, sector bends, quadrupoles, sextupoles, and octupoles, for a total of 39 elements, resulting in 58 parameters (Table I).[11] With exception of sector bends, all magnets are modeled as drift-kick-drift. For simplicity, we do not implement fringe fields or particle losses.

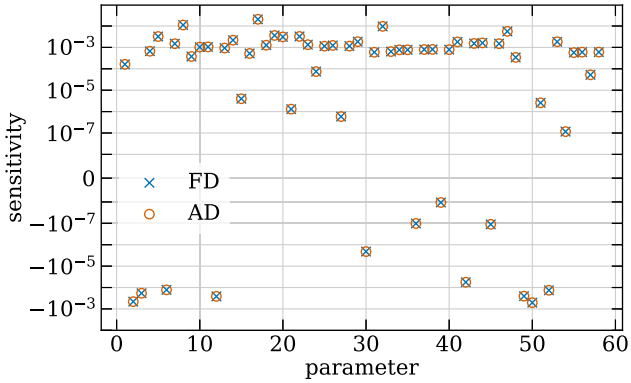


FIG. 7. Sensitivity of the objective function [Eq. (12)] with respect to beamline parameters, via autodifferentiation (AD) and finite differences (FD).

The objective is to minimize

$$\frac{1}{N_p}\sum_{i=1}^{N_p}(x_i^2 + y_i^2), \tag{12}$$

i.e., the mean distance from the center.

Figure 7 shows that the sensitivities of Eq. (12) with respect to beamline parameters obtained via autodifferentiation match those computed using (central) finite differences.

Having validated the numerical values, we now compare the time efficiencies of these methods. Figure 8 shows the execution times, absolute and normalized by the time to run one simulation, i.e., one primal. On CPU, AD-REV is one order of magnitude faster than FD. AD-FWD is also significantly slower than AD-REV, but still faster than FD. On GPU, AD-FWD is the fastest method for $N_p \leq 4000$ particles, above which it is AD-REV. On the other hand, FD is always slower than either AD-FWD or AD-REV and is the slowest for $N_p \geq 3000$. The speedup from using autodifferentiation ranges from marginal at $N_p = 1000$ to a factor of 3 at the upper end of the range of number of particles tested.

We remark that the lower computational complexity of the primal (no collective effects and lower number of slices) with respect to the cases in Sec. III A[12] is a factor that

[9]This estimate is obtained from JAX with `func.lower(x).compile().cost_analysis()`.

[10]Coherent synchrotron radiation is the only effect that could be considered, but for the purpose of this paper, it was considered negligible.

[11]The order of the elements and the values of the parameters of the initial design can be found in the project's repository.

[12]Confirmed by the primals' execution times [see Fig. 2(a) vs Fig. 8(a)].

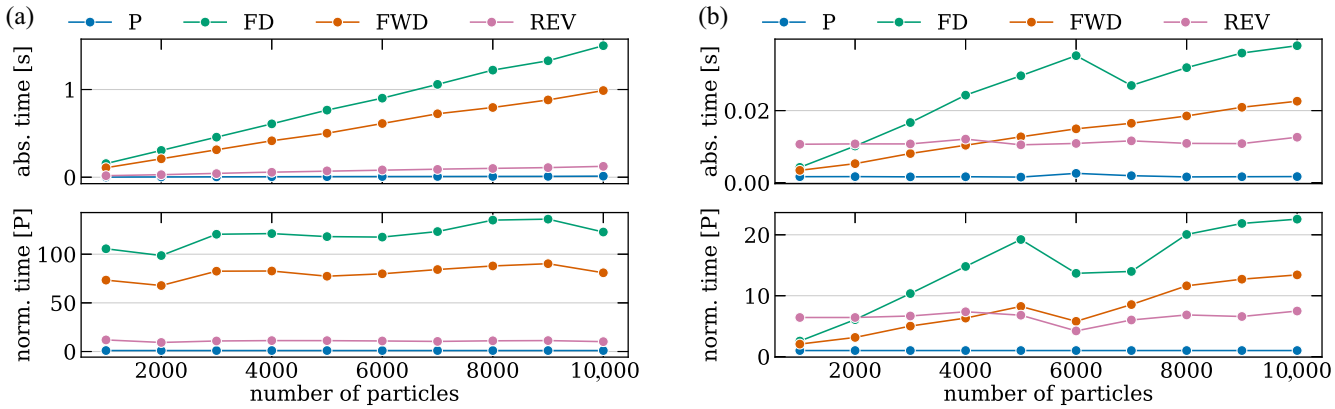


FIG. 8. AWAKE line—execution times. (a) CPU and (b) GPU.

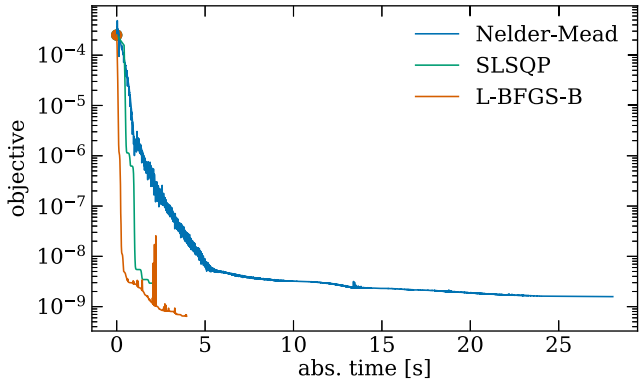


FIG. 9. Objective vs time for each of the methods.

benefits FD, relative to AD (see Sec. II B). Nevertheless, we observe the opposite: the overperformance of AD over FD increases. This can be explained by the increase of number of parameters from ≤ 30 to 58. This comparison highlights the two positive factors of overperformance of AD over FD: computational complexity of the primal and number of parameters.

Even if AD is more efficient than FD at computing the gradient, it does not necessarily follow that gradient-based optimization with AD is more efficient than gradient-free methods. Therefore, we compare gradient-based methods, SLSQP [23] and L-BFGS-B [24], with one of the most common and performant gradient-free methods, Nelder-Mead [6] (it dominated the gradient-free methods we tested). For a balance between a more realistic scenario (e.g., no nonsensical negative-length drifts) with simplicity, we bound the parameters to $\pm 50\%$ of their initial values. All optimizations start from the same design point. Figure 9 shows the evolution of each of the optimizations over time. Both gradient-based methods achieve their optima faster than the gradient-free method and their optima are better. Figure 10 shows the initial distribution and each of the three final distributions, showing the noticeable improvement in the objective.

IV. CONCLUSION

We analyzed, both theoretically and empirically, the efficiency of finite differences vs autodifferentiation, concluding that AD-REV is almost always faster than FD. The speedup can range from marginal to orders of magnitude and depends on the physical problem and the computational machine. It is more significant on CPU than on GPU, owing to the latter's parallel processing power, of which FD

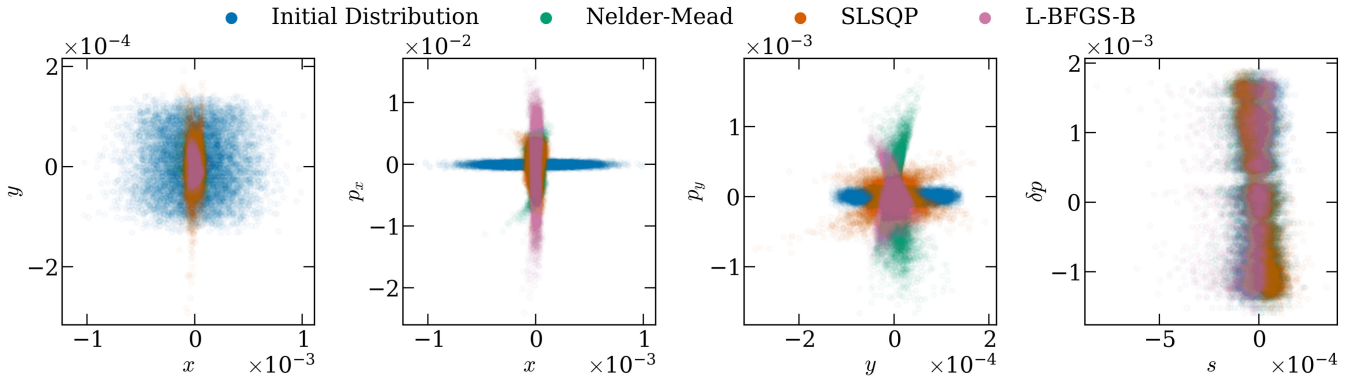


FIG. 10. Initial distribution and final distributions for each of the methods. The final distribution of the initial design is not shown because it is much larger than the plotting ranges.

can better take advantage. Notwithstanding, even in cases where FD is faster than AD-REV, it is slower than AD-FWD. Moreover, we embedded the sensitivities calculated via AD-REV in a gradient-based optimizer to solve a parameter-bounded problem, and compared that with a common gradient-free method, finding that the gradient-based approach is faster and reaches a better minimum.

This work shows that differentiable simulations present significant advantages in beamline optimization and have the potential to improve accelerator design in the search for higher and more robust performance.

DATA AVAILABILITY

The data that support the findings of this article are openly available [21].

APPENDIX A: AUTODIFFERENTIATION

1. Example

As explained in Sec. II C, autodifferentiation represents the numerous computational operations of a computationally-complex function as a directed acyclic graph (DAG) of, usually unary or binary, elementary operations. Figure 11 shows such a graph for the horizontal kick of a thin sextupole, which is given by the expression

$$\Delta x' = \frac{k_2}{2}(y^2 - x^2), \tag{A1}$$

where k_2 is the sextupole strength, and x and y are the horizontal and vertical positions. From left to right and in depth-first order, the expression can be broken down into:

$$u_1 = k_2, \tag{A2}$$

$$u_2 = 2, \tag{A3}$$

$$u_3 = u_1/u_2, \tag{A4}$$

$$u_4 = y, \tag{A5}$$

$$u_5 = u_4^2, \tag{A6}$$

$$u_6 = x, \tag{A7}$$

$$u_7 = u_6^2, \tag{A8}$$

$$u_8 = u_5 - u_7, \tag{A9}$$

$$u_9 = u_3 \times u_8, \tag{A10}$$

$$\Delta x' = u_9. \tag{A11}$$

This is shown in Fig. 11(a). With given values for k_2, x, and y, the graph can be traversed and $\Delta x'$ can be obtained at u_9. Then, starting from the top-most node, i.e., the quantity we wish to differentiate, reversed traversal is executed. At every node with result u_i, the partial derivative is computed with respect to each argument (i.e., for each child edge), a value that is then multiplied by the chained derivative to produce the chained derivative of the argument node, i.e., $\partial u_9/\partial u_i$. For example, the partial derivative of u_5 with respect to its argument, u_4, is $\partial u_5/\partial u_4 = 2u_4$, which is then multiplied by the chained derivative,

$$u_3 2u_4 = \left(\frac{\partial u_9}{\partial u_8}\frac{\partial u_8}{\partial u_5}\right)\frac{\partial u_5}{\partial u_4} = \frac{\partial u_9}{\partial u_4}. \tag{A12}$$

Figures 11(b) and 11(c) show these steps. The steps are shown in separate figures for clarity, but they are applied in

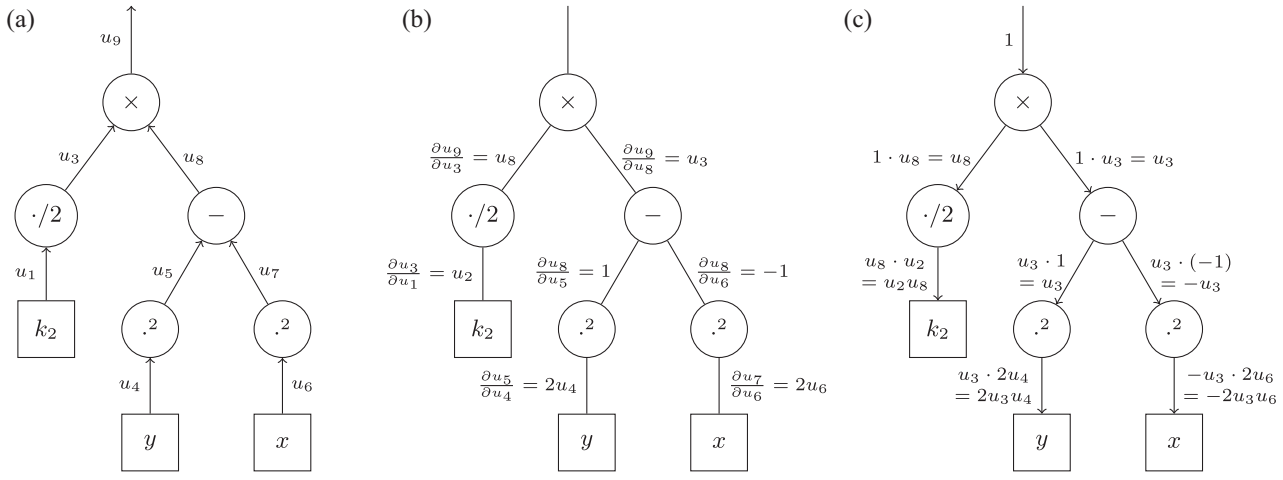


FIG. 11. Reverse autodifferentiation of the horizontal kick of a thin sextupole, $\frac{k_2}{2}(y^2 - x^2)$. For simplicity, $u_2 = 2$, the second argument in $k_2/2$, has been merged into the operation node. (a) Primal, (b) Differentiation, and (c) Accumulation.

succession at each edge of the graph. Finally, at the leaves, the expressions for the chained gradients are

$$\begin{pmatrix} \frac{\partial u_9}{\partial k_2} \\ \frac{\partial u_9}{\partial x} \\ \frac{\partial u_9}{\partial y} \end{pmatrix} = \begin{pmatrix} u_2 u_8 \\ -u_3 u_6 \\ u_3 u_4 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}(y^2 - x^2) \\ -k_2 x \\ k_2 y \end{pmatrix}, \tag{A13}$$

which match the analytical differentiation of Eq. (A12).

2. Cost analysis

The AD framework also presents an alternative way to derive the computational cost of differentiation. Here, as in most computational machines, we consider a DAG of unary or binary operations, although the same could be done for any arity. The first step is to compute the primal via a forward pass, which costs

$$C_{\mathrm{P}} = \sum_{i=1}^{n} c_{\mathrm{p}}^{(i)}, \tag{A14}$$

where $c_{\mathrm{p}}^{(i)}$ is the cost of the operation of node i and n is the number of nonleaf nodes.

The second step is backpropagation, where, for each nonleaf node, one must compute the derivative with respect to each of its children, multiply by the chained derivative and accumulate in the child node.

The derivatives cost

$$C_{\mathrm{D}} = \sum_{i=1}^{n} (c_{\mathrm{d}}^{(i,1)} + c_{\mathrm{d}}^{(i,2)}) \tag{A15}$$

$$= \sum_{i=1}^{n} \frac{c_{\mathrm{d}}^{(i,1)} + c_{\mathrm{d}}^{(i,2)}}{c_{\mathrm{p}}^{(i)}} c_{\mathrm{p}}^{(i)} \tag{A16}$$

$$= \sum_{i=1}^{n} k_{\mathrm{dp}}^{(i)} c_{\mathrm{p}}^{(i)} \tag{A17}$$

$$= \kappa_{\mathrm{dp}} \sum_{i=1}^{n} c_{\mathrm{p}}^{(i)} \tag{A18}$$

$$= \kappa_{\mathrm{dp}} C_{\mathrm{P}}, \tag{A19}$$

where $c_{\mathrm{d}}^{(i,j)}$ is the cost of computing the derivative of node i with respect to its jth child—while the ratio κ_{dp} depends on the operations of the primal, usually, the derivative has a similar cost to the function itself (e.g., $\sin \mapsto \cos$), i.e., normally $\kappa_{\mathrm{dp}} \sim O(1)$. The computation of the accumulated chained derivative costs one multiplication and addition per edge, i.e.,

$$C_{\mathrm{C}} = n_{\mathrm{e}} c_{\mathrm{ma}} \tag{A20}$$

$$= \left(\sum_{i=1}^{n} l_{\mathrm{node}}^{(i)} + \sum_{i=1}^{m} l_{\mathrm{param}}^{(i)} \right) c_{\mathrm{ma}} \tag{A21}$$

$$= (\bar{l}_{\mathrm{node}} n + \bar{l}_{\mathrm{param}} m) c_{\mathrm{ma}} \tag{A22}$$

$$= \frac{(\bar{l}_{\mathrm{node}} n + \bar{l}_{\mathrm{param}} m) c_{\mathrm{ma}}}{C_{\mathrm{P}}} C_{\mathrm{P}} \tag{A23}$$

$$= (\kappa_{\mathrm{sp}} + \kappa_{\mathrm{mp}} m) C_{\mathrm{P}}, \tag{A24}$$

where n_{e} is the number of edges, $l_{\mathrm{node}}^{(i)}$ and $l_{\mathrm{param}}^{(i)}$ are the number of parent edges of the internal nodes and the parameter nodes, and the bar denoting average. Adding up the three costs gives

$$\frac{C_{\mathrm{A}}}{C_{\mathrm{P}}} = \frac{C_{\mathrm{P}} + C_{\mathrm{D}} + C_{\mathrm{C}}}{C_{\mathrm{P}}} = 1 + \kappa_{\mathrm{dp}} + \kappa_{\mathrm{sp}} + \kappa_{\mathrm{mp}} m, \tag{A25}$$

which coincides with Eq. (11).

APPENDIX B: ACCELERATOR PHYSICS VS MACHINE LEARNING

A beamline can be seen as a neural network, where each beamline element is a network layer. Thus, it would seem that the remarkable efficiency of AD-REV in machine learning should transfer to beamline optimization. This is, alas, not necessarily the case. Comparing networks, those of machine learning are shallow (even in deep learning) relative to those in beamlines. At the same time, they are wider and more connected. For example, while one dense neural network layer may add tens to thousands of parameters, a quadrupole only adds two. Similarly, the dense layer adds tens to thousands of nonlinearities, while a sextupole only adds two.[13] For a beamline to have a significant number of parameters, such that using AD-REV is advantageous, it must be somewhat deep. Concomitantly, depth may be disadvantageous, as AD-REV requires keeping the graph of the operations of the primal in memory. This suggests that a jump in application of AD-REV from beamline to lattice, wherein hundreds to thousands of turns are executed, may not be as straightforward as it might seem at first sight—the depth of the DAG would be proportional to the lattice size and the number of turns. In this case, the judicious choice might potentially be AD-FWD, where the chained derivatives are computed and brought up along with the primal, foregoing the need to keep the graph in memory. Moreover, the results on GPU of Secs. III A and III B suggest that the

[13]It is perhaps then not surprising that one of the few publications [25] on application of AD to beamlines is on optimizing a neural network that generates the initial distribution, and not on optimizing the beamline elements.

considerable theoretical disadvantage of forward mode with respect to reverse may not be as significant in practice and may therefore not be an impediment to the application of AD-FWD.

In essence, while we show theoretically and empirically that AD-REV (and AD-FWD) can be (and usually is) faster than FD, it should not be assumed that, because it generally is so in other fields, it should also be in accelerator physics—while the methodology is similar, the specifics are not.

[1] C. Iselin, The MAD program, in *Computing in Accelerator Design and Operation* (Springer, Berlin, Heidelberg, 1984), 10.1007/3540139095_98.

[2] R. Ramjiawan, S. Döbert, J. Farmer, E. Gschwendtner, F. M. Velotti, L. Verra, G. Z. Della Porta, V. Bencini, and P. N. Burrows, Design and operation of transfer lines for plasma wakefield accelerators using numerical optimizers, Phys. Rev. Accel. Beams **25**, 101602 (2022).

[3] M. Remta, F. Velotti, F. Huhn, P. Arrutia Sota, and Y. Dutheil, Towards tailored beam distributions for fixed target experiments at CERN, in *Proceedings of the 16th International Particle Accelerator Conference, Taipei, Taiwan* (JACoW, Geneva, Switzerland, 2025), pp. 967–970, https://meow.elettra.eu/81/author/matthias-remta-cern-ch/index.html.

[4] L. Yang, D. Robin, F. Sannibale, C. Steier, and W. Wan, Global optimization of an accelerator lattice using multi-objective genetic algorithms, Nucl. Instrum. Methods Phys. Res., Sect. A **609**, 50 (2009).

[5] L. Deniau *et al.*, MethodicalAcceleratorDesign/MAD-X: 5.09.01, 2023, 10.5281/zenodo.10255261.

[6] J. A. Nelder and R. Mead, A simplex method for function minimization, Comput. J. (United Kingdom) **7**, 308 (1965).

[7] G. Iadarola *et al.*, Xsuite: An integrated beam physics simulation framework, in *Proceeding of the 68th ICFA Advanced Beam Dynamics Workshop on High-Intensity and High-Brightness Hadron Beams (HB2023), Geneva, Switzerland* (JACoW, Geneva, Switzerland, 2024), TUA2I1, 10.18429/JACoW-HB2023-TUA2I1.

[8] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík, *Comparing Results of 31 Algorithms from the Black–Box Optimization Benchmarking BBOB–2009* (Association for Computing Machinery, New York, NY, USA, 2010), pp. 1689–1696.

[9] A. Jameson, Aerodynamic design via control theory, J. Sci. Comput. **3**, 233 (1988).

[10] L. Magri, Adjoint methods as design tools in thermoacoustics, Appl. Mech. Rev. **71**, 020801 (2019).

[11] Our World in Data, Computation used to train notable AI systems, by domain, https://ourworldindata.org/grapher/ai-training-computation-vs-parameters-by-domain (2024).

[12] J. Ansel *et al.*, *PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation* (Association for Computing Machinery, New York, NY, USA, 2024), pp. 929–947.

[13] M. Abadi *et al.*, TensorFlow, large-scale machine learning on heterogeneous systems, 2015, https://zenodo.org/records/16852354.

[14] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, JAX: Composable transformations of Python+NumPy programs, 2025, http://github.com/jax-ml/jax.

[15] J. Gonzalez-Aguilera, Y.-K. Kim, R. Roussel, A. Edelen, and C. Mayes, Towards fully differentiable accelerator modeling, in *Proceedings of the 14th International Particle Accelerator Conference (IPAC23), Venice, Italy* (JACoW, Geneva, Switzerland, 2023), pp. 2797–2800, 10.18429/JACoW-IPAC2023-WEPA065.

[16] J. Kaiser, C. Xu, A. Eichler, and A. Santamaria Garcia, Bridging the gap between machine learning and particle accelerator physics with high-speed, differentiable simulations, Phys. Rev. Accel. Beams **27**, 054601 (2024).

[17] J. Wan, H. Alamprese, C. Ratcliff, J. Qiang, and Y. Hao, JuTrack: A Julia package for auto-differentiable accelerator modeling and particle tracking, Comput. Phys. Commun. **309**, 109497 (2025).

[18] L. Deniau, MAD-NG, a standalone multiplatform tool for linear and non-linear optics design and optimisation, arXiv:2412.16006.

[19] J. Wan, H. Alamprese, C. Ratcliff, J. Qiang, and Y. Hao, JuTrack code repository, 2025, https://github.com/MSU-Beam-Dynamics/JuTrack.jl [Accessed 2025-06-01].

[20] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, Automatic differentiation in machine learning: A survey, J. Mach. Learn. Res. **18**, 1 (2018).

[21] F. Huhn and F. M. Velotti, ADPT–Auto-Differentiable Particle Tracking, 2025, 10.5281/zenodo.17305665.

[22] J. Qiang, Differentiable self-consistent space-charge simulation for accelerator design, Phys. Rev. Accel. Beams **26**, 024601 (2023).

[23] D. Kraft, Algorithm 733: TOMP–Fortran modules for optimal control calculations, ACM Trans. Math. Softw. **20**, 262 (1994).

[24] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization, ACM Trans. Math. Softw. **23**, 550 (1997).

[25] R. Roussel, J. P. Gonzalez-Aguilera, E. Wisniewski, A. Ody, W. Liu, J. Power, Y.-K. Kim, and A. Edelen, Efficient six-dimensional phase space reconstructions from experimental measurements using generative machine learning, Phys. Rev. Accel. Beams **27**, 094601 (2024).