ACCELERATOR OPTIMIZATIONS USING SciBmad WITH PyTorch

C. Ung[1], M. G. Signorelli[1,2], G. H. Hoffstaetter[1,2], D. C. Sagan[1]
[1]CLASSE, Cornell University, Ithaca, NY, USA
[2]Brookhaven National Laboratory, Upton, NY, USA

Abstract

SciBmad is a new, fully differentiable software ecosystem for accelerator physics, usable in Julia and/or Python. Full differentiability enables efficient and numerically-reliable optimizations of particle accelerators, in particular with machine learning (ML) methods. One of the most commonly used ML frameworks is PyTorch, which provides powerful, versatile, and straightforward tools for such purposes. We implemented PyTorch bindings with SciBmad, enabling seamless integration of SciBmad's powerful and differentiable simulations with a standard PyTorch workflow. With these bindings, standard gradient-based PyTorch optimization workflows can be used seamlessly with SciBmad's differentiable simulations. In this paper, we describe the implementation details and present some examples that demonstrate SciBmad-PyTorch workflows.

INTRODUCTION

Gradient based optimization and machine learning (ML) methods are increasingly applied in accelerator physics, but require efficient and accurate derivatives of beam dynamic simulations. In practice, these derivatives are typically computed through finite differences which scale poorly with system size and introduce numerical errors sensitive to step size. Automatic Differentiation (AD) addresses both limitations by applying the chain rule directly to the sequence of operations defining a program, providing exact derivatives at a cost comparable to a single function evaluation.

SciBmad is a new, fully differentiable accelerator physics simulation framework usable in both Julia and Python, featuring CPU/GPU parallelized 6d symplectic integrators. Unlike existing tools that treat simulation and differentiation as separate concerns, SciBmad's polymorphic design makes it intrinsically differentiable, allowing AD directly through simulation without subsequent instrumentation [1].

Julia's Just-In-Time (JIT) compilation allows high-level flexibility while retaining low-level performance and utilizes a dynamic typing system that enables rapid prototyping, which is essential in scientific development [2]. Julia's multiple dispatch system is crucial in the context of AD and ML. This feature allows function behavior to be defined based on argument types, allowing type-generic implementations that accommodate the specialized numerical types required by different AD modes without modifying the underlying code [3, 4]. Together, these features make SciBmad a high-performance, natively differentiable simulation environment that enables modern ML workflows directly within accelerator physics.

Existing accelerator simulation tools typically rely on finite differences for gradient estimation and lack straightforward integration with modern ML frameworks. SciBmad addresses this with a fast, type-generic, and differentiable simulation [5]. However, the majority of ML and optimization workflows in the scientific community are developed in Python. PyTorch has become the dominant platform for gradient based methods due to its dynamic computation graphs, broad ecosystem, and active development community [6]. To make SciBmad work seamlessly within PyTorch-based optimization workflows, we developed Python bindings that allow gradients computed through SciBmad's differentiable simulations to be passed directly to PyTorch's optimization system, enabling researchers to leverage SciBmad's full differentiable physics stack from within a familiar Python environment.

BASIC USAGE

Upon importing the SciBmad module, Julia-defined SciBmad simulation functions are available within a standard PyTorch workflow as native Python callables. The following example displays the construction of a FODO cell and defines a tracking function:

Listing 1: Beamline construction and particle tracking in SciBmad-PyTorch interface

```
import SciBmad as sb
import torch

torch.set_default_dtype(torch.float64)

qf = sb.Quadrupole(Kn1=0.36, L=0.5)
d = sb.Drift(L=1.2)
qd = sb.Quadrupole(Kn1=-0.36, L=0.5)

fodo = sb.Beamline(
    [qf, d, qd, d],
    E_ref=18e9,
    species_ref=sb.Species("electron"),
)

def foo(v, k1):
    qf.Kn1 = k1
    qd.Kn1 = -k1
    res = sb.track(fodo, v0=v)
    return res.v[0, :, :]

coords0 = torch.tensor([5e-3, 1e-3, 5e-3, -1e-3,
     0.0, 0.0])

with torch.no_grad():
    target = foo(coords0, torch.tensor(10.0))[:4,
     -1]

k1 = torch.tensor(5.0, requires_grad=True)
```

```
optimizer = torch.optim.LBFGS([k1], lr=0.5, max_iter
    =1, line_search_fn="strong_wolfe")

for i in range(2):
    def closure():
        optimizer.zero_grad()
        result = foo(coords0, k1)[:4, -1]
        loss = (result - target).pow(2).sum()
        loss.backward()
        return loss

    loss = optimizer.step(closure)

    with torch.no_grad():
        result = foo(coords0, k1)[:4, -1]
        true_loss = (result - target).pow(2).sum()
```

Gradients are computed through Julia's AD backend and returned to PyTorch transparently.

For physics models beyond the built-in library, arbitrary Julia functions can be introduced at runtime via `sb.define()` and used identically within the optimization loop. The following example defines a Legendre polynomial in Julia and optimizes its parameters to approximate a target function entirely from Python:

Listing 2: Legendre P3 optimization with SciBmad + PyTorch

```
sb.define("""
function legendre_p3(x)
    if x isa AbstractArray
        return 0.5 .* (5 .* x.^3 .- 3 .* x)
    else
        return 0.5 * (5 * x^3 - 3 * x)
    end
end
""")

optimizer = torch.optim.SGD([a, b, c, d], lr=5e-6)

for i in range(2000):
    optimizer.zero_grad()
    y_pred = a + b * sb.legendre_p3(c + d * x)
    loss = (y_pred - y).pow(2).sum()
    loss.backward()
    optimizer.step()
```

The user-defined function participates in the same differentiable execution path as any built-in SciBmad element with no additional configuration[1].

DESIGN

The SciBmad PyTorch binding connects Python-based ML workflows with Julia-based accelerator physics simulations through a module proxy pattern. The standard Python SciBmad module is replaced at import time by a custom object that intercepts attribute access and dynamically resolves functions from Julia's main namespace. For most users, package setup is handled automatically given a compatible Python and Julia installation. When a Julia function is accessed, it is wrapped in a callable that inspects arguments at runtime. `torch.Tensor` arguments are routed through a differentiable execution path while plain Python or Numpy values call Julia directly.

A custom function, `torch.autograd.Function` [7], ensures differentiability. In the forward pass, input PyTorch tensors are converted to a Numpy array and forwarded to Julia for simulation. In the backward pass, the appropriate Julia AD jacobian is selected based on input and output dimensionality (Table 1). The resulting gradients are injected into PyTorch's computational graph.

Table 1: Selection of Julia AD Based on Input and Output Dimensionality

Input	Output	Jacobian Type
Scalar	Scalar	`derivative`
Vector	Scalar	`gradient`
Vector	Vector	`jacobian`
Scalar	Vector	`jacobian` (scalar wrapped in length-1 vector)

The mechanism enabling AD to pass through the simulation without modification is Julia's type promotion system. Every tracking function calls `promote_type` across all inputs and casts everything to the promoted type "T" before constructing the element and calling `track!`. Under standard execution, "T" resolves to Float64. However, when ForwardDiff creates a dual number, `promote_type` lifts all inputs to Dual(...) and the physics computation proceeds on dual arithmetic, carrying the calculated value and derivatives through the simulation automatically. Data exchange between runtimes is handled through explicit tensor to Numpy, Numpy to Julia conversion at each boundary. This introduces modest overhead while eliminating memory aliasing and cross-runtime type inconsistencies.

Beamline elements are constructed as typed Python objects with named parameters (for example: `sb.Quadrupole(Kn1=..., L=...)` and assembled into an `ObjectBeamline` via `sb.Beamline()`, which carries reference energy, species, and element sequence. Tracking is performed through `sb.track()` which propagates coordinates sequentially through each element. Since each element call passes through the same proxy and autograd machinery, PyTorch's computational graph is built incrementally across the full sequence. A complete multi-element beamline is therefore differentiable end to end with no additional instrumentation besides specifying `requires_grad = True`. Beyond the built-in element library, arbitrary Julia functions can be introduced at runtime through `sb.define()`, after which they are immediately accessible as differentiable PyTorch compatible callables.

RESULTS

In a real accelerator, quadrupole field strengths deviate from their nominal design values due to manufacturing tolerances and installation errors. These deviations are not directly measurable but affect the beam trajectory. To

[1] User-defined Julia functions must branch on `x isa AbstractArray` to handle both scalar and vector inputs, as PyTorch may pass either depending on context. Built-in SciBmad elements handle this internally.

demonstrate the binding as a practical optimization tool, a FODO lattice is constructed with a focusing and defocusing quadrupole pair whose strengths are constrained symmetrically as `qf.Kn1 = k1` and `qf.Kn1 = -k1`.A target coordinate state is computed by tracking a particle at a known reference strength, and the optimizer is initialized at a perturbed value with no knowledge of the target. It is then tasked with recovering the correct strength by minimizing the mean squared error between the tracked and observed coordinates without any prior knowledge of the true error values, using a standard `PyTorch` optimizer. Gradients are computed via Julia AD through the binding with no finite differences at any stage.

Table 2: Convergence of quadrupole strength recovery optimization. Target values are $k_{n1} = [5.1, -4.9]$.

Iteration	Loss	k_{n1}[0]	k_{n1}[1]
0	3.519×10^{-8}	5.00096	-4.99902
100	2.053×10^{-9}	5.07610	-4.92327
200	1.430×10^{-11}	5.09802	-4.90191
300	6.320×10^{-15}	5.09996	-4.90004
400	7.390×10^{-21}	5.10000	-4.90000

The optimizer recovers target quadrupole strengths to machine precision within 500 iterations, with loss decreasing from 3.52×10^{-8} to below 10^{-20}. The result (Table 2) confirms that the full sequence (beamline construction, particle tracking, Julia AD, and `PyTorch` parameter updates) operates correctly as a seamless end-to-end differentiable workflow from Python.

BENCH MARKING

Benchmarks were performed comparing Julia AD through the binding against finite differences in Python, following a full JIT warmup phase, median timing is reported throughout (Table 3).

For a single element loss, Julia AD through the binding computes gradients in 0.124 ms compared to 0.573 ms for finite differences, this is a 4.61× speedup. For a full 10-element beamline the speedup is 4.69× (2.466 ms vs 11.57 ms). AD requires only 2 functions per gradient step regardless of parameter count while finite differences require 1 + 2p evaluations for p number parameters. Gradient accuracy was assessed via relative L2 error: 1.871×10^{-13} for a single element and 9.785×10^{-8} for full beamline, both attributable to finite difference truncation error. AD gradients are exact up to floating point arithmetic.

Table 3: Gradient computation time: Julia AD through binding vs finite differences in Python.

Configuration	Julia AD Median (µs)	FD Median µs	Speedup
Single element	124.4	573.1	4.61×
Full beamline (10 elements)	2466.2	11576.5	4.69×

Table 4 shows a fixed step gradient descent comparison on the FODO matching problem. Both methods converge to the same final loss for 1.0039×10^{-6}, confirming equivalent optimization quality. AD achieves this in 200 function evaluations versus 1300 for finite differences. This is a 6.5× reduction in simulation calls for identical convergence. In a fixed-time comparison over approximately 1 second, AD completed 276 steps against 72 for finite differences, reaching relatively the same final loss while utilizing fewer total function evaluations as shown in Table 5.

Table 4: Fixed-step Optimization Comparison Over 100 Iterations

Method	Final Loss	Steps	Evals
GD + Julia AD	$1.003\,988 \times 10^{-6}$	100	200
GD + FD	$1.003\,988 \times 10^{-6}$	100	1300

Table 5: Fixed-time Optimization Comparison Over Approximately 1 s

Method	Final Loss	Steps	Evals
GD + Julia AD	$1.003\,702 \times 10^{-6}$	276	552
GD + FD	$1.004\,033 \times 10^{-6}$	72	936

CONCLUSION

A python interface to the `SciBmad` accelerator physics simulation framework has been presented, enabling seamless integration of Julia-based beam dynamics simulations with `PyTorch` based optimization workflows. The binding is implemented through a proxy module that transparently wraps Julia functions as differentiable `PyTorch` callables, requiring no modification to the underlying physics implementation and no Julia knowledge on the part of the user. Automatic differentiation through Julia's `DifferentiationInterface.jl` is exposed to `PyTorch`'s autograd system through a custom `torch.autograd.Function`. This enables exact gradient computation through beam dynamics simulations at a fraction of the cost of finite differences. Benchmarking demonstrates an approximate 4.6x speedup in gradient computation over finite differences across both single element and full beamline configurations with equal optimization convergence quality. A quadrupole strength recovery demonstration confirms that the full pipeline (beam construction, particle tracking, Julia AD, and `PyTorch` parameter updates) operates correctly and converges to machine precision.

Future work includes support for reverse-mode AD, which would improve efficiency for large-scale lattice optimization problems, and extension to multi-particle bunch tracking to broaden applicability to collective effects and beam distribution optimization. Overall, the presented binding makes `SciBmad`'s high-performance differentiable accelerator physics directly accessible within standard gradient-based `PyTorch` workflows, enabling gradient-based optimization of beam dynamics simulations without sacrificing the performance or numerical accuracy of the underlying physics simulation.

REFERENCES

[1] M. G. Signorelli *et al.*, "SciBmad: A differentiable and GPU-parallelized software library for particle accelerator design, nonlinear analysis, and machine learning", presented at the 17th Int. Part. Accel. Conf. (IPAC'26), Deauville, France, May 2026, paper THP5325, this conference.

[2] S. Pal *et al.*, "A next-generation dynamic programming language Julia: Its features and applications in biological science", *J. Adv. Res.*, vol. 64, pp. 143–-154, Oct. 2024. doi:10.1016/j.jare.2023.11.015

[3] "Automatic differentiation", *Wikipedia, The Free Encyclopedia*, Accessed 01. May 2026: https://en.wikipedia.org/wiki/Automatic_differentiation

[4] "Finite difference method", *Wikipedia, The Free Encyclopedia*, Accessed 01. May 2026: https://en.wikipedia.org/wiki/Finite_difference_method

[5] M. Signorelli and D. Sagan, "SciBmad: A full-featured ecosystem for modern, differentiable accelerator physics simulations", presented at the 14th Int. Comput. Accel. Phys. Conf. (ICAP'24), Seeheim-Jugenheim, Germany, Oct. 2024. https://indico.gsi.de/event/19249/contributions/82658/attachments/48832/70947/SciBmad.pdf

[6] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library", *arXiv*, 2019. doi:10.48550/arXiv.1912.01703

[7] "PyTorch: Defining New autograd Functions", *PyTorch Tutorials*, Accessed 01. May 2026: https://docs.PyTorch.org/tutorials/beginner/examples_autograd/polynomial_custom_function.html