ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC25-WEBN2

A MODULE FOR FAST AUTO DIFFERENTIABLE SIMULATIONS*

J. Qiang[1†], Y. Hao[2], A. Qiang[3], J. Wan[2]
[1]Lawrence Berkeley National Laboratory, Berkeley, [2]Michigan State University, East Lansing,
[3]Stanford Univeristy, Stanford, USA

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2024). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

Abstract

The auto differentiable simulation is a type of simulation that outputs of the simulation include not only the simulation result itself, but also their derivatives with respect to various input parameters. It provides an efficient method to study sensitivity of the simulation results with respect to the input parameters. Furthermore, it can be used in gradient based optimization methods for rapidly optimizing design parameters. In this paper, we present the development of a fast auto-differentiation module designed for integration into numerous simulation codes.

INTRODUCTION

Recent studies within the particle accelerator community have shown significant interest in auto-differentiable simulation [1–6]. Unlike conventional simulation methods, differentiable simulation not only yields the desired outcomes but also computes their derivatives with respect to various given parameters during the simulation process. These parameters can encompass initial charged particle beam characteristics or the accelerator lattice control and acceleration settings employed in the simulation.

The derivatives obtained from differentiable simulations offer quantitative insights into the sensitivity of the simulation results to these parameters. Such sensitivities are valuable for establishing tolerance limits for the corresponding machine parameters in accelerator design. Moreover, these derivatives can be utilized in gradient-based optimization algorithms, such as conjugate gradient optimization, to facilitate rapid accelerator design parameter optimization.

Automatic differentiation is a computational technique that allows for the calculation of derivatives of complex functions with respect to their variables by applying a set of fundamental function rules. This technique avoids the need for symbolic derivative representations or numerical approximations. Instead, it breaks down the function evaluation into a series of elementary operations and simple functions, using the chain rule to combine the derivatives of these basic components to determine the overall function derivatives.

It operates in two primary modes: forward mode and reverse mode. For a composite function, forward mode applies the chain rule from left to right (or inside to outside), whereas reverse mode applies it from right to left (outside to inside).

Auto-differentiation has found extensive application in the artificial intelligence/machine learning (AI/ML) community for training neural networks through gradient-based optimization [7]. Several AI/ML frameworks, including PyTorch [8] and TensorFlow [9], incorporate this functionality.

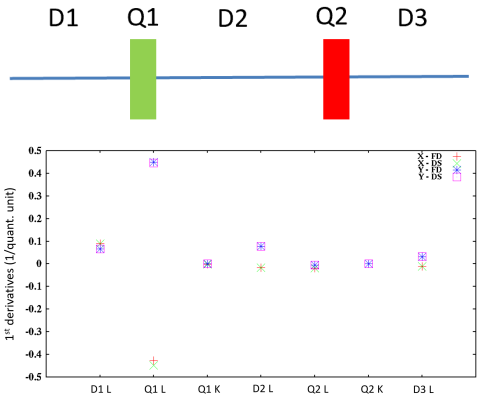

Figure 1: Derivatives of final horizontal and vertical emittances with respect to seven lattice parameters from the single differentiable self-consistent space-charge simulation and from the finite difference approximation to the first derivative [4].

While the forward and reverse modes of automatic differentiation can be implemented using tree structures in AI/ML, the particle accelerator community has traditionally used a method called truncated power series algebra (TPSA) to compute high-order transfer maps in beam dynamics studies [10]. This method has also been recently applied to differentiable space-charge simulation [4].

Figure 1 illustrates the derivatives of the final horizontal and vertical emittances of a proton beam with respect to seven lattice parameters of a drift, focusing, drift, defocusing, drift (FODO) lattice. These derivatives were obtained from a single differentiable self-consistent space-charge simulation and compared with the first derivative approximated using finite difference methods with eight simulation results. The excellent agreement between these approaches highlights the accuracy of auto-differentiation, which yields exact results within computer precision, unlike the finite difference method that is subject to numerical truncation errors. In this example, a Fortran version of the high-order TPSA library was employed to compute the derivatives [11]. However, using the entire high-order TPSA library for first-order derivative computation is inefficient and time-consuming; in the aforementioned example, it took approximately 860 seconds on a supercomputer.

To address this efficiency challenge, we developed a fast module specifically for auto-differentiable simulation. This

* Work supported by the U.S. DOE under Contract No. DE-AC02-05CH11231.
† jqiang@lbl.gov

ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC25-WEBN2

new module achieved the same result in approximately 4 seconds for the example above, representing a speedup of over 200 times compared to the standard TPSA library.

IMPLEMENTATION

Our fast module adopts the TPSA method but applies it exclusively to the first derivative. This significantly simplifies the arithmetic operations and the implementation within the module. This approach is also known as the dual number implementation of forward mode auto-differentiation in the AI/ML community [12].

In the TPSA method, calculating a function's derivatives with respect to its variables is transformed into evaluating the function of a vector-like variable according to specific algebraic rules. For the first derivative in our module, we define a TPSA vector variable F where the first element is the function value, and the subsequent elements are the partial derivatives with respect to each variable of interest: $F = (f, f_{x_1}, f_{x_2}, \cdots, f_{x_n})$. Here, f is the function value, and f_{x_i} represents the partial derivative with respect to variable x_i. For an individual variable x_i, its corresponding vector is $X_i = (x_i, 0, 0, \cdots, 1, \cdots)$, where 1 is located at the $(i+1)^{th}$ element.

The addition rule for two vector variables F and G is defined as:

$$F + G = (f + g, f_{x_1} + g_{x_1}, f_{x2} + g_{x_2}, \cdots, f_{x_n} + g_{x_n}) \quad (1)$$

The multiplication rule is:

$$FG = (fg, gf_{x_1} + fg_{x_1}, gf_{x_2} + fg_{x_2}, \cdots, gf_{x_n} + fg_{x_n}) \quad (2)$$

The division rule is:

$$\frac{F}{G} = (\frac{f}{g}, \frac{f_{x_1}g - g_{x_1}f}{g^2}, \frac{f_{x_2}g - g_{x_2}f}{g^2} \cdots, \frac{f_{x_n}g - g_{x_n}f}{g^2}) \quad (3)$$

The function of a vector variable $h(F)$ can be represented as:

$$h(F) = (h(f), h_f f_{x_1}, h_f f_{x_2}, \cdots, h_f f_{x_n}) \quad (4)$$

where h_f denotes the partial derivative of h with respect to f, i.e., $\partial h / \partial f$.

Based on these operational rules, we can define a specialized data type for this vector, along with its corresponding operators and common special functions. Figure 2 shows a section of the C++ implementation of this module's class definition.

APPLICATION EXAMPLES

In the following, we will present several application examples for the above fast auto differentiation module. Figure 3 shows a test example to compute a multi-variable function $f(x_1, x_2, x_3, x_4, x_5) = x_3(2cos(x_1/x_2) + x_1/x_2 + e^{x_2}) + 2x_4 + sinh(x_4) + x_5^2$ and its derivative with respect to these five variables at variable space $(3, 0.1, 0.1, 1, 1)$ location. The function value and its derivatives are also printed out. In order to use the fast AD module, we need to declare the

```
class TPSAad{
public:
  static const int dimmax = 5;
  double map[dimmax+1] = {0.0,};
  int terms = dimmax+1;
  const TPSAad operator+(const TPSAad &M);
  const TPSAad operator-(const TPSAad &M);
  const TPSAad operator*(const TPSAad &M);
  const TPSAad operator/(const TPSAad &M);
  friend TPSAad exp(const TPSAad &);
  friend TPSAad log(const TPSAad &);
  friend TPSAad sqrt(const TPSAad &);
  friend TPSAad sin(const TPSAad &);
  friend TPSAad cos(const TPSAad &);
  friend TPSAad tan(const TPSAad &);
  ......
};
```

Figure 2: A section of the auto differentiation module class definition in C++.

```
#include "TPSAad.h"
#include <iostream>

int main(){
  TPSAad x1,x2,x3,x4,x5,func;

  x1.assign(3.0,1);
  x2.assign(0.1,2);
  x3.assign(0.1,3);
  x4.assign(1.0,4);
  x5.assign(1.0,5);

  //test function
  func = (2*cos(x1/x2)+x1/x2+exp(x2))*x3+2*x4+sinh(x4) + pow(x5,2.0);

  cout<<"func. value and its derivatives w.r.t. 5 variables"<<endl;
  cout<<func.map[0]<<" "<<func.map[1]<<" "<<func.map[2]<<" "<<
   func.map[3]<<" "<<func.map[4]<<" "<<func.map[5]<<endl;
}
```

Figure 3: A C++ code showing the usage of the AD class module.

function name and the five variable names as objects of the AD class. The expression of the function form stays the same. This shows that to use the AD module in a computer program, we can keep most part the program except replacing the original data type of the function and the variables that one would like to compute derivatives with this new data type.

In the second example, we examined how the Twiss parameters $(\beta_x, \alpha_x, \beta_y, \alpha_y)$ change after passing through the FODO channel illustrated in Fig. 1. We specifically assessed the sensitivity of these parameters with respect to seven key FODO lattice parameters: the lengths of the first drift space (D1 L), the first quadrupole (Q1 L), the focusing strength of the first quadrupole (Q1 k), the length of the second drift space (D2 L), the length of the second quadrupole (Q2 L), the focusing strength of the second quadrupole (Q2 k), and

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2024). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC25-WEBN2

© Content from this work may be used under the terms of the CC BY 4.0 licence (© 2024). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

the length of the third drift space (D3 L). For this analysis, the normalized quadrupole strengths were set to 22 m^{-2} and $-22\ m^{-2}$, respectively. We defined the relative change of a Twiss parameter as $\frac{xdT}{Tdx}$, where T represents β_x, α_x, β_y, or α_y, and x is the lattice parameter being varied. The results obtained from our auto-differentiable simulation showed excellent agreement with those calculated using the central finite difference approximation (as shown in Fig. 4). Our findings indicate that the beta function values exhibit the highest sensitivity to the length and strength of the quadrupoles.

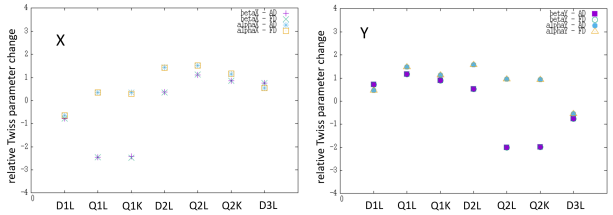

Figure 4: Relative Twiss parameter changes with respect to 7 lattice parameters from a single auto differentiable simulation and from central finite difference approximation with 14 simulations.

The derivatives from the differentiable simulation can be used with a gradient-based parameter optimizer for accelerator lattice control parameter optimization. In the third example, we integrated the differentiable simulation module with a conjugate gradient optimizer to attain the quadrupole strengths inside a matching section in front of a periodic FODO lattice. A schematic plot of the matching section lattice and the periodic FODO lattice is shown in Fig. 5. It consists of a periodic FODO lattice, a quadrupole matching section, and another periodic FODO lattice. The four quadrupoles in the matching section were used to match the given Twiss parameters at the entrance to the second periodic FODO lattice. The Polak-Ribiere conjugate gradient optimization method [13] was used to minimize the objective function that is defined as follows:

$$f(\mathbf{k}) = \frac{(\beta_x(\mathbf{k}) - \beta_{xt})^2}{\beta_{xt}^2} + (\alpha_x(\mathbf{k}) - \alpha_{xt})^2 + \frac{(\beta_y(\mathbf{k}) - \beta_{yt})^2}{\beta_{yt}^2} + (\alpha_y(\mathbf{k}) - \alpha_{yt})^2 \quad (5)$$

where $\mathbf{k}$ is a set of control variables, α_{xt}, β_{xt}, α_{yt}, and β_{yt} are the target Twiss parameters at the entrance to the periodic lattice, and the α_x, β_x, α_y, and β_y are the beam Twiss parameters computed from the differentiable simulation. These Twiss parameters depend on the focusing strengths of the quadrupoles inside the matching section. These strengths are control variables in the above objective function. Using the above differentiable simulation module, the first derivatives of the objective function with respect to the four control variables were obtained in addition to the objective function value. These derivatives are used to construct a direction conjugate to the gradient direction to guide the search for the minimum solution. Figure 6 shows the evolution of the proton beam's transverse RMS size as it passes through the

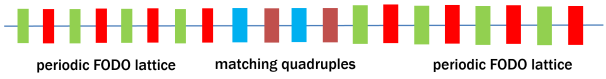

Figure 5: Schematic plot of two periodic FODO lattices and a section of matching section between them.

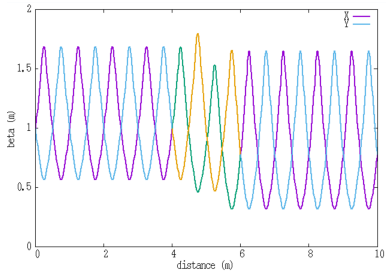

Figure 6: Evolution of Twiss beta function value through the above lattice after optimal matching.

FODO lattice. As can be seen in the figure, the beta function transitions smoothly from one periodic FODO lattice to another, achieved by optimizing the strengths of the four matching quadrupoles.

We have implemented the above fast auto differentiation module using a variety of programming languages: Fortran, C++, Java, Python, and Julia. To evaluate the performance of these different implementations, we measured the computational time required for a single particle to track through 1000 turns of a lattice composed of 100 periods of FODO cells. Figure 7 illustrates the computing time as a function

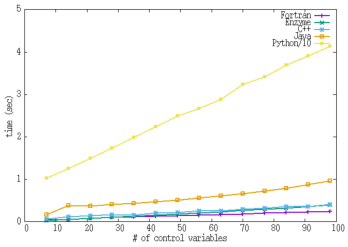

Figure 7: Computing time as a function of number of control variables with Fortran, C++, Java, Python, and Enzyme library implemenation.

of the number of control variables for the Fortran, C++, Java, and Python implementations. For comparison, we also included the computing time obtained using the well-known Enzyme package [14]. As the figure shows, the Fortran implementation exhibited the best performance, closely followed by the C++ implementation, which demonstrated comparable performance to the Enzyme library. This is perhaps expected, given Fortran's highly optimized nature for scientific computing. The Python implementation showed the slowest performance, likely due to its interpreted nature, which necessitates runtime translation.

ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC25-WEBN2

REFERENCES

[1] R. Roussel, A. Edelen, D. Ratner, K. Dubey, J. P. Gonzalez-Aguilera, Y.K. Kim, and N. Kuklev, "Differentiable Preisach modeling for characterization and optimization of particle accelerator systems with hysteresis", *Phys. Rev. Lett.*, vol. 128, p. 204801, 2022. doi:10.1103/PhysRevLett.128.204801

[2] R. Roussel and A. L. Edelen, "Applications of differentiable physics simulations in particle accelerator modeling", *arXiv*, 2022. doi:10.48550/arXiv.2211.09077

[3] R. Roussel, A. Edelen, C. Mayes, D. Ratner, J. P. Gonzalez-Aguilera, S. Kim, E. Wisniewski, and J. Power, "Phase space reconstruction from accelerator beam measurements using neural networks and differentiable simulations", *Phys. Rev. Lett.*, vol. 130, p. 145001, 2023. doi:10.1103/PhysRevLett.130.145001

[4] J. Qiang, "Differentiable self-consistent space-charge simulation for accelerator design", *Phys. Rev. Accel Beams*, vol. 26, p. 024601, 2023. doi:10.1103/PhysRevAccelBeams.26.024601

[5] J. Kaiser, C. Xu, A. Eichler, and A. S. Garcia, "Bridging the gap between machine learning and particle accelerator physics with high-speed, differentiable simulations", *Phys. Rev. Accel Beams*, vol. 27, 054601, 2024. doi:10.1103/PhysRevAccelBeams.27.054601

[6] J. Wan, H. Alamprese, C. Ratcliff, J. Qiang, and Y. Hao, "JuTrack: a Julia package for auto-differentiable accelerator modeling and particle tracking", *Comp. Phys. Comm.*, vol. 309, p. 109497, 2025. doi:10.1016/j.cpc.2024.109497

[7] C. C. Margossian, "A Review of Automatic Differentiation and its Efficient Implementation", *WIREs Data Min. Knowl. Discov.*, vol. 9, no. 4, p. e1305, 2019. doi:10.1002/widm.1305

[8] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library", in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035.

[9] Martin Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems", 2015. https://www.tensorflow.org/

[10] M. Berz, "Differential Algebraic Description of Beam Dynamics to Very High Orders", *Part. Accel.*, vol. 24, pp. 109–124, 1989.

[11] Y. Hao, https://github.com/YueHao/PyTPSA

[12] https://en.wikipedia.org/wiki/Automatic_differentiation

[13] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in Fortran: the Art of Scientific Computing*. New York:Cambridge University Press, 1992.

[14] W. Moses and V. Churavy, "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients", in *Advances in Neural Information Processing Systems*, 2020, pp. 12472–12485. https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf,

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2024). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.