DIFFERENTIABLE PROGRAMMING FOR PARTICLE PHYSICS SIMULATIONS

R. Grinis[*] [**]

Moscow Institute of Physics and Technology
141700, Dolgoprudny, Moscow Region, Russia

Received August 24, 2021,
revised version August 24, 2021
Accepted for publication October 6, 2021

DOI: 10.31857/S0044451022020043

Abstract. We describe how to apply adjoint sensitivity methods to backward Monte-Carlo schemes arising from simulations of particles passing through matter. Relying on this, we demonstrate derivative based techniques for solving inverse problems for such systems without approximations to underlying transport dynamics. We are implementing those algorithms for various scenarios within a general purpose differentiable programming C++17 library NOA (github.com/grinisrit/noa).

1. Overview of the main results. In this paper, we explore the challenges and opportunities that arise in integrating differentiable programming (DP) with simulations in particle physics.

In our context, we will broadly refer to DP as a program for which some of the inputs could be given the notion of a variable, and the output of that program could be differentiated with respect to them.

Most common examples include the widely used deep learning (DL) models created over the powerful automatic differentiation (AD) engines such as TensorFlow and PyTorch. Since their initial release, those machine learning (ML) frameworks grew up into fully-fledged DP libraries capable of tackling a more diversified set of tasks.

Recently, a very fruitful interaction between DP as we know it in ML and numerical solutions to differential equations started to gather pace with the work of Chen et al. [1]. A whole new area tagged now-days Neural Differential Equations arose in scientific ML.

On one hand, using ML we obtain a more flexible framework with a wealth of new tools to tackle a variety of inverse problems in mathematical modeling. On the other hand, many techniques in the latter such as the adjoint sensitivity methods give rise to new powerful algorithms for AD.

A few implementations are now available:

— torchdiffeq is the initial python package developed by [1] providing ODE solvers that not only integrate with PyTorch DL models, but also use those to describe the dynamics;

— torchsde builds off from torchdiffeq and provides the same functionality for SDEs, as well as $\mathcal{O}(1)$-memory gradient computation algorithms, see [2];

— diffeqflux is a Julia package developed by Rackauckas et al. [3] and relies on a rich scientific ML ecosystem treating many different types of equations including PDEs.

Unsurprisingly, one can also find roots of this story in computational finance, see for example the work of Giles et al. [4]. An AD algorithm is presented there for computing the risk sensitivities for a portfolio of options priced through Monte-Carlo simulation. That set-up is close to our case of interest and therefore represents a great source of inspiration for us.

In fact, for particle physics simulations a similar picture is left almost unexplored so far. The dynamics are richer than the ones considered before, but we also

[*] E-mail: roland.grinis@grinisrit.com
[**] GrinisRIT ltd., London, UK

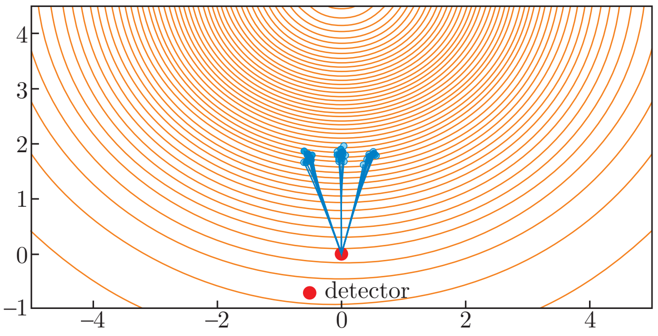

Fig. 1. (Color online) This is a toy example. The contours correspond to level sets of the materials mixture given by a Gaussian centered at $\vartheta_\mu = (0,5)$ with scale $\vartheta_\sigma = \sqrt{10}$. In blue we show the BMC simulated trajectories. For the sake of simplicity, we assume the known particle flux is constant equals to one and is reached after two steps. A naive implementation of the BMC scheme to compute the flux in this configuration can be found in the Appendix, routine backward_mc 1.There is, however, a challenge with the approach consisting in differentiating through the MC simulation with AD. The algorithm is not scalable in the number of steps for the discretisation of the transport. This issue can be addressed by adjoint sensitivity methods. The routine backward_mc_grad 2 in the Appendix provides an implementation with first order derivative for the BMC scheme in this example

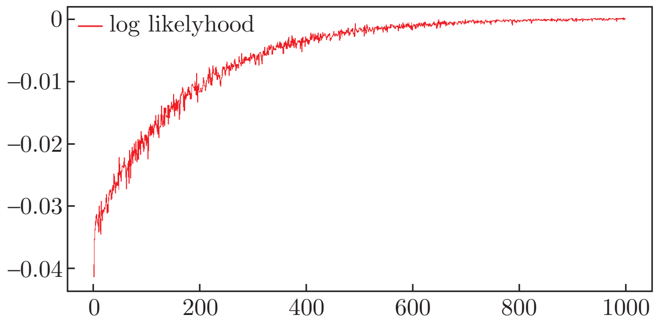

Fig. 2. After implementing BMC over LibTorch's Autograd library we can use a gradient based optimisation to solve the inverse problem for ϑ_μ and ϑ_σ. Let us set $\vartheta_\mu = (-1,5)$ arbitrarily. We present SGD convergence over 1000 steps with learning rate 0.05, reaching $\hat{\vartheta}_\mu = (-1.0033, 4.9891)$ and $\hat{\vartheta}_\sigma^2 = 9.9611$ in good agreement with true values

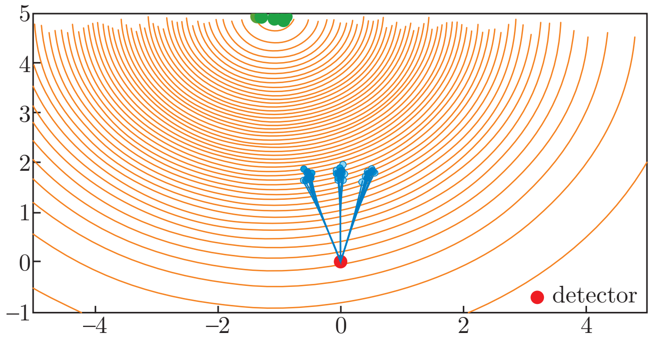

Fig. 3. (Color online) Optimal parameters and Bayesian posterior sampled using NOA [8] implementation of Riemannian HMC with an explicit symplectic integrator

have more tools at our disposal such as the Backward Monte-Carlo (BMC) techniques [5, 6]. We make use of the latter to adapt the adjoint sensitivity methods [7] to the transport of particles through matter simulations.

Ultimately, we obtain a novel methodology for image reconstruction problems when the absorption mechanism is non-linear. In future, one can demonstrate this approach in the specific case of muography. We are releasing our implementations within the open source library NOA [8].

The results are shown in Figs. 1–3.

2. Conclusion. In this paper, we have demonstrated how to efficiently integrate automatic differentiation and adjoint sensitivity methods with BMC schemes arising in the passage of particles through matter simulations.

We believe that this builds a whole new bridge between scientific machine learning and inverse problems arising in particle physics. In future, we hope to prove the success of this technique in a variety of image reconstruction problems with non-linear dynamics, starting with muography.

Acknowledgments. I would like to thank the MIPT–NPM lab and A. Nozik in particular for very fruitful discussions that have led to this work. We are very grateful to GrinisRIT for the support.

This work has been initially presented at the QUARKS-2020 workshop — serie "Advanced Computing in Particle Physics".

The full text of this paper is published in the English version of JETP.

Appendix. Code examples. We have collected here the two BMC implementations for our basic example. You can reproduce all the calculations in this paper from the notebook differentiable_programming_pms.ipynb available in NOA [8].

For the specific code snippets here the only dependency is LibTorch:

```
#include <torch/torch.h>
```

The following routine will be used throughout and provides the rotations by a tensor angles for multiple scattering:

```
inline torch::Tensor rot(const torch::Tensor &angles)
{
const auto n = angles.numel();
const auto c = torch::cos(angles);
const auto s = torch::sin(angles);
return torch::stack({c, -s, s, c}).t().view({n, 2, 2});
}
```

Given the set-up in Fig. 1 example we define:

```
const auto detector = torch::zeros(2);
const auto materialA = 0.9f;
const auto materialB = 0.01f;

inline const auto PI = 2.f * torch::acos(torch::tensor(0.f));

inline torch::Tensor mix_density(
const torch::Tensor &states,
const torch::Tensor &vartheta)
{
return torch::exp(-(states - vartheta.slice(0, 0, 2))
.pow(2).sum(-1) / vartheta[2].pow(2));
}
```

Example 1. This implementation relies completely on the AD engine for tensors. The whole trajectory is kept in memory to perform reverse-mode differentiation.

The routine accepts a tensor theta representing the angles for the readings on the detector, the tensor node encoding the mixture of the materials which is essentially our variable, and the number of particles npar.

It outputs the simulated flux on the detector corresponding to theta:

```
inline torch::Tensor backward_mc(
const torch::Tensor &theta,
const torch::Tensor &node,
const int npar)
{
const auto length1 = 1.f - 0.2f * torch::rand(npar);
const auto rot1 = rot(theta);

auto step1 = torch::stack({torch::zeros(npar), length1}).t();
step1 = rot1.matmul(step1.view({npar, 2, 1})).view({npar, 2});
const auto state1 = detector + step1;

auto biasing = torch::randint(0, 2, {npar});
auto density = mix_density(state1, node);
auto weights =
torch::where(biasing > 0,
(density / 0.5) * materialA,
((1 - density) / 0.5) * materialB) *
torch::exp(-0.1f * length1);

const auto length2 = 1.f - 0.2f * torch::rand(npar);
const auto rot2 = rot(0.05f * PI * (torch::rand(npar) - 0.5f));
auto step2 =
length2.view({npar, 1}) * step1 / length1.view({npar, 1});

step2 = rot2.matmul(step2.view({npar, 2, 1})).view({npar, 2});
const auto state2 = state1 + step2;

biasing = torch::randint(0, 2, {npar});
density = mix_density(state2, node);
weights *=
torch::where(biasing > 0,
(density / 0.5) * materialA,
((1 - density) / 0.5) * materialB) *
torch::exp(-0.1f * length2);

// assuming the flux is known equal to one at state2
return weights;
}
```

Example 2. This routine adopts the adjoint sensitivity algorithm to earlier Example 1. It outputs the value of the flux and the first order derivative w.r.t. the tensor node:

```
inline std::tuple<torch::Tensor, torch::Tensor> backward_mc_grad(
const torch::Tensor &theta,
const torch::Tensor &node)
{
const auto npar = 1; //work with single particle
auto bmc_grad = torch::zeros_like(node);

const auto length1 = 1.f - 0.2f * torch::rand(npar);
const auto rot1 = rot(theta);
auto step1 = torch::stack({torch::zeros(npar), length1}).t();
step1 = rot1.matmul(step1.view({npar, 2, 1})).view({npar, 2});
const auto state1 = detector + step1;

auto biasing = torch::randint(0, 2, {npar});
auto node_leaf = node.detach().requires_grad_();
auto density = mix_density(state1, node_leaf);
auto weights_leaf = torch::where(biasing > 0,
(density / 0.5) * materialA,
((1 - density) / 0.5) * materialB) * torch::exp(-0.01f * length1);

bmc_grad += torch::autograd::grad({weights_leaf}, {node_leaf})[0];
auto weights = weights_leaf.detach();

const auto length2 = 1.f - 0.2f * torch::rand(npar);
const auto rot2 = rot(0.05f * PI * (torch::rand(npar) - 0.5f));
auto step2 = length2.view({npar, 1}) * step1 / length1.view({npar, 1});
step2 = rot2.matmul(step2.view({npar, 2, 1})).view({npar, 2});
const auto state2 = state1 + step2;

biasing = torch::randint(0, 2, {npar});
node_leaf = node.detach().requires_grad_();
density = mix_density(state2, node_leaf);
weights_leaf = torch::where(biasing > 0,
(density / 0.5) * materialA,
((1 - density) / 0.5) * materialB) * torch::exp(-0.01f * length2);

const auto weight2 = weights_leaf.detach();
bmc_grad = weights * torch::autograd::grad({weights_leaf}, {node_leaf})[0]
+ weight2 * bmc_grad;
weights *= weight2;

// assuming the flux is known equal to one at state2
return std::make_tuple(weights, bmc_grad);
}
```

REFERENCES

1. R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, in *Proc. Advances in Neural Information Processing Systems 31*, pp. 6571–6583 (2018).

2. X. Li, T.-K. L. Wong, R. T. Q. Chen, and D. K. Duvenaud, *23^d International Conference on Artificial*

Intelligence and Statistics, Proc. Machine Learning Res. **108**, 2677 (2020).

3. C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, and A. Ramadhan, arXiv:2001.04385.

4. L. Capriotti and M. B. Giles, *Algorithmic Differentiation: Adjoint Greeks Made Easy*, SSRN Electronic Journal (2011).

5. L. Desorgher, F. Lei, and G. Santin, Nucl. Instrum. Meth. A **621**, 247 (2010).

6. V. Niess, A. Barnoud, C. Carloganu, and E. Le Menedeu, Comput. Phys. Comm. **229**, 54 (2018).

7. L. S. Pontryagin, E. F. Mishchenko, V. G. Boltyanskii, and R. V. Gamkrelidze, *The Mathematical Theory of Optimal Processes*, John Wiley S., New York, London (1962).

8. Differentiable Programming for Optimisation Algorithms over LibTorch, https://github.com/grinisrit/noa.