Slight Stochastic Shifts Suffice: Cross-Trajectory Vectorized Estimation of Simulation Gradients

Philipp Andelfinger
Nanyang Technological University
Singapore, Singapore
philipp.andelfinger@ntu.edu.sg

Wentong Cai
Nanyang Technological University
Singapore, Singapore
aswtcai@ntu.edu.sg

Abstract

Monte Carlo gradient estimators enable an efficient gradient-driven local search in a simulation's parameter space. Partial derivatives are estimated based on the simulation outputs at random perturbations around the current parameter combination. However, the effective computational cost grows with the number of perturbations. Here, we explore the use of modern CPUs' vector instructions to reduce the estimation time on a single processor core. We vectorize across simulation trajectories, based on the hypothesis that the perturbations can be chosen small enough that the control flow divergence remains low. Control flow is realized using a predication scheme, allowing model code to remain similar to its scalar counterpart. Since the approach trivially benefits numerical simulations without parameter-dependent control flow, our evaluation instead considers the calibration of a building evacuation model in which transitive effects of perturbations change the neighborhood relation among pedestrians. Our cross-trajectory vectorization scheme speeds up the model's calibration via simulation-based inference by a factor of about 1.5 without occupying additional cores.

CCS Concepts

• **Computing methodologies** → **Agent / discrete models**; • **Mathematics of computing** → **Numerical differentiation**.

Keywords

Agent-based simulation, gradient estimation, vectorization

ACM Reference Format:
Philipp Andelfinger and Wentong Cai. 2025. Slight Stochastic Shifts Suffice: Cross-Trajectory Vectorized Estimation of Simulation Gradients. In *39th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '25), June 23–26, 2025, Santa Fe, NM, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3726301.3728410

1 Introduction

The ability of gradient descent for swift local search over high-dimensional input spaces is a key factor for the successes in deep learning. Building on results from infinitesimal perturbation analysis [21], the past years have seen a renewed interest in extending the reach of gradient descent to various types of simulations to accelerate calibration, simulation-based inference, the search for optimal system designs, and reinforcement learning. At the core of these efforts is the handling of discrete jumps such as those caused by conditional branching, which are not captured by gradients computed in a straightforward manner via automatic differentiation (AD) [14]. Broadly, the solutions fall into two categories: *AD-based approaches* directly evaluate the derivative expressions corresponding to the mathematical operations involved in the simulation, but support only certain forms of control flow [3, 4] or require problem-specific tuning [1, 11]. *Finite difference-type estimators* determine gradients based on the changes in the simulation output when varying the parameters either deterministically or stochastically. While estimators such as REINFORCE [22] can be specialized to produce unbiased estimates, artificial random perturbations allow for approximate estimations for arbitrary simulations.

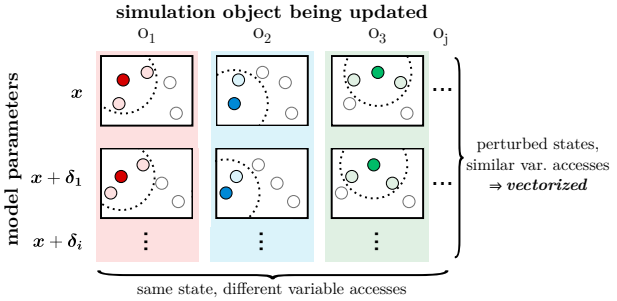


Figure 1: Snapshots of pedestrian simulations under different parameter perturbations. The neighbor sets (dotted circles) vary among simulation objects (o_i), but remain largely (here: exactly) the same across replications. We thus vectorize across replications to accelerate the ensemble.

Both classes of estimators sample the simulation output, making overall optimization processes time-consuming. Our work is motivated by the observation that the replications within such an ensemble are likely to follow similar control flow paths. A classical technique to exploit shared state trajectories by avoiding recomputations is simulation cloning [10, 13]. However, in continuous state spaces, stochastic parameter perturbations can cause states to diverge quickly, even on identical control flow paths. Then, cloning quickly degenerates to executing the replications independently. Similarly, caching of previously encountered states [20] is unlikely to succeed if most simulation objects' exact states are unique.

The idea behind our approach is to vectorize across perturbed replications. An idealized example is shown in Figure 1: pedestrians (circles) move in a two-dimensional continuous space. Their state updates depend on nearby pedestrians' states, restricted by an interaction radius (dotted circle). The neighborhood relation determines the operands involved in each state update. At a given time, this relation is different for each pedestrian, and hence their

SIGSIM-PADS '25, Santa Fe, NM, USA

ACM ISBN 979-8-4007-1591-4/25/06
https://doi.org/10.1145/3726301.3728410

state updates will access different sets of state variables. On the other hand, the neighborhood relation in the example is unaffected by the slight perturbations, making it possible to efficiently vectorize each individual pedestrian's state update across all replications. Importantly, different from existing approaches for parallelizing across state updates or replications [4, 5, 12, 18], our use of modern CPUs' vector instructions accelerates the ensemble runs without using additional processor cores, although parallelization can still be applied to further decrease execution times.

We make the following contributions:

(1) We present a simple abstraction that vectorizes mostly unmodified model code across replications. A predication scheme transparently handles divergent control flow paths.
(2) The execution time reductions and the maximum degree of tolerable perturbations are assessed in an inference problem over an evacuation simulation. Optimization curves show the practical benefits of the approach.

2 Cross-Trajectory Vectorization

We aim to make use of vector instructions to accelerate ensembles of similar simulation runs on a single processor core. Recent CPUs offer vector instructions for a variety of mathematical, logical, and memory operations, typically with vector widths of 256 bits as part of the AVX2 instruction set extension, or 512 bits using AVX512. Hence, these instructions apply to up to 16 floating point numbers in single precision, or 8 numbers in double precision.

Key to our approach are vectorized data types that allow users to operate on state variables across all replications at the same time. To this end, we offer the type `vfloat`, which represents a floating point variable with operations applying to multiple replications. Trivially, `vfloat2` extends the type to 2-dimensional spaces.

In the following, we discuss the main challenge of handling divergent control flow across replications, particularly when the divergence changes the set of variables involved in state updates.

2.1 Handling Divergent Branches

If a parameter-dependent control flow path is not taken by all replications, the subsequent operations must only be applied to a subset of replications. To achieve this, we describe a vectorized imperative branching construct `vec_if` as shown in Algorithm 1. Two key aspects are the handling of nested branching, and the avoidance of unnecessary overhead by skipping branches not taken by any replication. The global list `conds` holds per-replication truth values at the current branch nesting level, initially set to all-ones. On variable assignments, the tail element of `conds` decides which replications the assignment applies to by masking all inactive replications. The boolean `any_true` indicates whether there is any active replication. When we encounter a `vec_if` statement and the condition is false for all replications, we set `any_true` to false and skip the branch body. In this case, `vec_fi` resets `any_true` to account for outer branch levels. If, instead, the branch condition is true for any replication, we compute the logical `AND` between the previous and current branch levels. If any condition remains true, we append all truth values to `conds` and execute the branch body. Subsequently, `vec_fi` removes the last element from `conds`. This vectorization scheme is reminiscent of the "single instruction, multiple data" execution abstraction offered by graphics processing units, in which divergent "threads" are serialized in a similar manner [6].

Algorithm 1 Translation of vectorized `if`-construct via predication. Branches not taken by any replication are skipped.

vec_if COND: ⋮ **[if body]** →	**if any**(COND): new_conds ← conds[-1] ∧ COND any_true ← **any**(new_conds) **if** any_true: conds.**append**(new_conds) **else**: any_true ← **False** **if** any_true: ⋮ **[if body]**
vec_fi →	**if** any_true: conds.**pop**() any_true ← **True**

Algorithm 2 Sample user code (left) and first loop iterations (right).

1:	**vfloat v1[#objects], v2[#objects]**	`conds: [(1,1)]`
	⋮ **[initialization]**	`v1: [`o_1`: (1,0), `o_2`: ...]`
		`v2: [`o_1`: (0,1), `o_2`: ...]`
2:	**for** step **in** 1, #steps:	
3:	**for** o **in** 1, #objects:	
4:	**vec_if** v1[o] > 0:	`conds: [(1,1); (1,0)]`
5:	v2[o] += 1	`v2: [`o_1`: (1,1), `o_2`: ...]`
6:	**vec_if** v2[o] > 0:	`conds: [(1,1); (1,0); (1,0)]`
7:	v2[o] += 1	`v2: [`o_1`: (2,1), `o_2`: ...]`
8:	**vec_fi**	`conds: [(1,1); (1,0)]`
9:	**vec_fi**	`conds: [(1,1)]`

In Algorithm 2, we show a snippet of sample user code and the first loop iterations in an example execution with two replications. Initially, all conditions in `conds` are true. On the outer `vec_if` (line 4), the condition is found to be true for the first replication, but false for the second, which is recorded in `conds`. Accordingly, the assignment `v2[o_1] += 1` is applied only for the first replication. The condition of the inner `vec_if` (line 6) is true for both replications. However, as the second replication's value in `conds` is false, it is again masked out in the branch body. Finally, the program exits the two branch levels, removing the tail items from `conds`.

2.2 Effects of Divergent Neighborhoods

Our approach is particularly relevant to agent-based simulations, as the dynamic neighborhood relations may prevent traditional vectorization across agents. Still, the more the neighborhoods diverge among replications, the fewer vectorization opportunities remain.

To provide an intuition about the permissible amount of divergence, we study the execution of an idealized neighbor-dependent update step. The scalar ensemble carries out $a_s = \sum_{r=1}^{R} n_r$ neighbor accesses, n_r being the number of neighbors in replication r. With vectorization, the effective number of accesses a_v is the number of unique neighbors that occur in *any* replication, i.e., up to $N-1$ in a simulation with N agents. Let us first consider the best case of identical neighborhoods in all replications. The effective number of neighbor accesses across the original scalar ensemble is $R \cdot n$, where

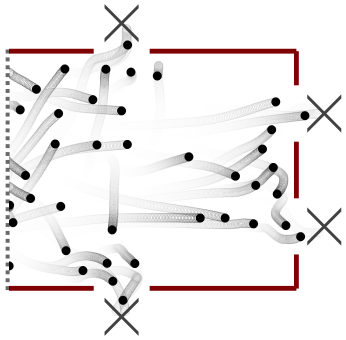


Figure 2: Screenshot of the evacuation simulation. The buildings' occupants (•) enter the scenario from the left and escape the building at a randomly chosen waypoint (×).

R is the number of replications and n the (here constant) number of neighbors per replication. By vectorization using a sufficient vector width, this is reduced by a factor of R to only n accesses.

The worst case occurs when each possible neighbor appears in exactly one replication, resulting in $a_s = a_v$. As we will see in Section 3, when including all overheads, the best case speedup factor is still substantially smaller than the vector width. Hence, to attain a benefit, it is decisive that the perturbations are small enough not to generate excessively diverse neighborhoods.

2.3 Implementation

We implemented vectorized arithmetic, logic, and control flow in the form of a wrapper around Fastor, a tensor library for C++ [17]. Conditional branching via predication is implemented by the preprocessor macros `vec_if` and `vec_fi`, which insert C++ code to keep track of the branch conditions' truth values across replications on each nesting level. Using Fastor's masking functionality, our custom types' overloaded operators are dynamically applied only to the replications whose predicates are true. Our implementation's source code is publicly available[1].

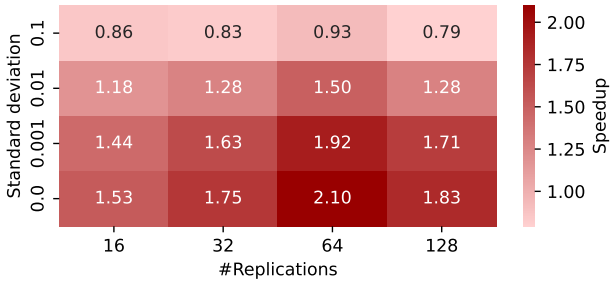
3 Experiments

We evaluate our vectorization scheme both for individual gradient estimates and in an overall optimization task. The key issue at hand is the trade-off between minimizing the execution time overhead, which favors small perturbations, and maximizing the optimization progress per step, which may favor large perturbations.

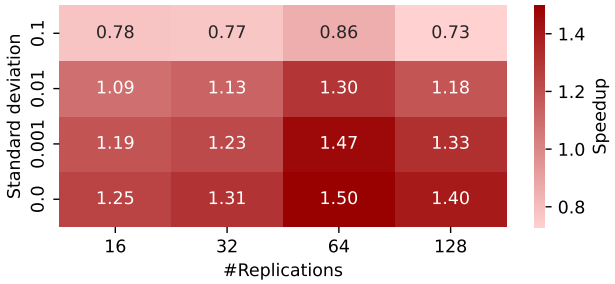
We employ Polyak's gradient oracle [16], which obtained low-variance estimates in similar tasks [2, 11]. For R replications including a reference run at parameters θ, the estimator is $\nabla f(\theta) \approx (R-1)^{-1} \sum_{r=1}^{R-1} \big(f(\theta + X_r) - f(\theta)\big) {X_r}^{-1}$ with $X_r \sim N(0, \sigma)$. We note that the speedup by vectorization depends on the perturbation size set via σ, not on the specific gradient estimator.

Execution times were averaged over 10 repetitions on an Intel Xeon W-2133 CPU with AVX512 and clang 18.1.3. The optimization task were averaged over 100 optimization processes per perturbation size on an AMD EPYC 7742 CPU with AVX2 and clang 18.1.8.

[1] https://github.com/philipp-andelfinger/VecSimGradPADS25



(a) Single precision.



(b) Double precision.

Figure 3: Speedup by vectorization depending on the number of simulation replications and the standard deviation of the input perturbations. The largest benefit of vectorization is seen with small perturbations and when using single-precision floating point numbers.

3.1 Simulation Model

We simulate the building evacuation scenario on the $30m \times 30m$ space shown in Figure 2. The building's occupants enter the scenario from the left at a rate of one per second and escape through randomly chosen exits. The simulation ends after 120s. Distance-keeping among pedestrians and from walls is handled by Helbing's Social Force model [9] applied at 0.1s intervals using leap frog integration. The simulation was implemented in a traditional scalar fashion and, with only minor changes, using our vectorized data types and the `vec_if` construct. The main computational cost stems from computing the forces from each pedestrians' neighbors, which we reduce in both implementation by neighbor lookups from a grid based on an interaction range of 5m. The conditional branches handled by our `vec-if` construct include checks whether a pedestrian is active, whether the target exit has been reached, as well as several branches to normalize angles among pedestrians and to obstacles.

We set up a simulation-based inference experiment similar to [8] by drawing the pedestrians' desired velocities from a two-component Gaussian mixture specified using six simulation parameters, representing the components' weights, means, and standard deviations. The pedestrians' evacuation times are evaluated with reference to a pre-defined five-component Gaussian target mixture as a placeholder representing empirical data. To calibrate the parameters via maximum likelihood estimation, the simulation output is chosen as the negative log-likelihood representing the goodness-of-fit.

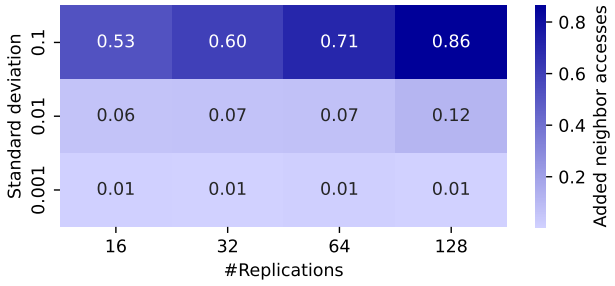


Figure 4: Average number of added neighbor accesses to account for divergent neighborhoods. More severe perturbations and larger numbers of replications cause more accesses.

3.2 Performance Measurements

The main influencing factors for the speedup by vectorization are the number of replications, the floating point precision, and the sparsity in the operations, which depends on the perturbation size. In Figure 3, we observe that as expected, the speedup is higher in single precision and with small perturbations, reaching values of 2.1 and 1.5 without control flow divergence. Speedups larger than 1 are observed up to a standard deviation of 0.01. Surprisingly, the speedup begins to slightly decrease beyond 64 replications, even without perturbations. In microbenchmarks, we observed a significant variation in speedup with different vector widths, which we speculate depends on clang's internal optimization heuristics.

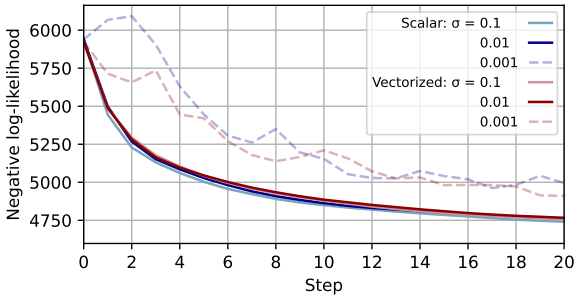
To examine the model-dependent effects of the perturbation size, we study the average number of additional neighbor accesses introduced by the perturbations, which dominate the execution time. As discussed in Section 2.2, the best-case speedup gives an indication of how many additional neighbor accesses can be tolerated. Figure 4 shows that with 64 replications, the perturbations with a standard deviation of 0.1 introduce about 0.8 additional accesses on average. Considering the best-case speedups of 2.1 and 1.5, a sharp decrease of the speedup is in line with expectations.

3.3 Simulation-Based Inference

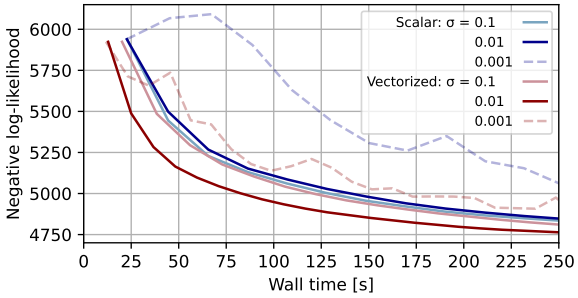
We now turn to an overall gradient-based optimization process to examine whether the small perturbations required for fast ensemble runs suffice to permit fast optimization progress. Figure 5 shows the progress over the gradient descent steps and over wall-clock execution time using single-precision floating point numbers. Our first observation is that the progress with a standard deviation of 0.001 is excessively slow and erratic. With the standard deviations of 0.01 and 0.1, there is no significant difference in the progress per step. However, with vectorization, as each step is substantially accelerated, the progress over wall-time is substantially faster. In line with the previous measurements, the vectorized optimization with standard deviation 0.01 attains any given log-likelihood about 1.5 times faster than the scalar runs.

4 Conclusions

In our case study, our vectorization scheme accelerated simulation-based inference by a factor of 1.5, without using additional processor cores. The benefits depend on the magnitude of the perturbations



(a) Progress over optimization steps.



(b) Progress over execution time.

Figure 5: Gradient-based inference over the evacuation simulation. While the improvement per step (a) is roughly identical in the scalar and vectorized case, the progress over wall time (b) is substantially accelerated by vectorization.

required for effective gradient descent and on the resulting control flow divergence, both of which are model dependent. We consider our approach best suited for models that contain only limited parameter-dependent control flow, but that do not allow for more straightforward vectorization within each trajectory (e.g., cellular automata [7]). An important field where such models appear is operations research. For instance, in inventory management problems [19], the model parameters may affect the availability of certain products over time, while the demand is parameter-independent.

Beyond experimentation on a wider range of models, avenues for future work include refinements to the vectorization approach. First, all operations are currently executed using vector instructions, even if a particular operation is required only by a single replication. Sparsity could be exploited by falling back to scalar instructions dynamically. Second, our implementation requires intermediate results within sequences of operations to be stored explicitly. Using source-to-source translation in place of the current operator overloading scheme, this overhead could be avoided without complicating the user code. Third, the approach can be combined with parallel execution across cores, e.g., in the form of a coarse-grained parallelization of independent optimization runs to increase the probability of converging to high-quality local optima [15].

Acknowledgments

This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG3-RP-2022-031).

References

[1] Philipp Andelfinger. 2023. Towards differentiable agent-based simulation. *ACM Transactions on Modeling and Computer Simulation* 32, 4 (2023), 1–26.

[2] Philipp Andelfinger and Justin N Kreikemeyer. 2024. Automatic gradient estimation for calibrating crowd models with discrete decision making. In *International Conference on Computational Science*. Springer, 227–241.

[3] Gaurav Arya, Moritz Schauer, Frank Schäfer, and Christopher Rackauckas. 2022. Automatic differentiation of programs with discrete randomness. *Advances in Neural Information Processing Systems* 35 (2022), 10435–10447.

[4] Ayush Chopra, Jayakumar Subramanian, Balaji Krishnamurthy, and Ramesh Raskar. [n. d.]. AgentTorch: agent-based modeling with automatic differentiation. In *Second Agent Learning in Open-Endedness Workshop*.

[5] Nicholson Collier and Michael North. 2011. Repast HPC: A Platform for Large-Scale Agent-Based Modeling. *Large-Scale Computing* (2011), 81–109.

[6] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. 2012. GPGPU processing in CUDA architecture. *arXiv preprint arXiv:1202.4347* (2012).

[7] Michael J Gibson, Edward C Keedwell, and Dragan A Savić. 2015. An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware. *J. Parallel and Distrib. Comput.* 77 (2015), 11–25.

[8] Marion Gödel, Nikolai Bode, Gerta Köster, and Hans-Joachim Bungartz. 2022. Bayesian inference methods to calibrate crowd dynamics models for safety applications. *Safety science* 147 (2022), 105586.

[9] Dirk Helbing and Peter Molnar. 1995. Social force model for pedestrian dynamics. *Physical review E* 51, 5 (1995), 4282.

[10] Maria Hybinette and Richard M Fujimoto. 2001. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 11, 4 (2001), 378–407.

[11] Justin N Kreikemeyer and Philipp Andelfinger. 2023. Smoothing methods for automatic differentiation across conditional branches. *IEEE Access* (2023).

[12] Georg Kunz, Daniel Schemmel, James Gross, and Klaus Wehrle. 2012. Multi-level parallelism for time-and cost-efficient parallel discrete event simulation on GPUs. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 23–32.

[13] Xiaosong Li, Wentong Cai, and Stephen J Turner. 2017. Cloning agent-based simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 27, 2 (2017), 1–24.

[14] Charles C Margossian. 2019. A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 9, 4 (2019), e1305.

[15] Rafael Martí, Mauricio GC Resende, and Celso C Ribeiro. 2013. Multi-start methods for combinatorial optimization. *European Journal of Operational Research* 226, 1 (2013), 1–8.

[16] Yurii Nesterov and Vladimir Spokoiny. 2017. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics* 17, 2 (2017), 527–566.

[17] Roman Poya, Antonio J. Gil, and Rogelio Ortigosa. 2017. A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Computer Physics Communications* (2017). https://doi.org/10.1016/j.cpc.2017.02.016

[18] Paul Richmond, Robert Chisholm, Peter Heywood, Mozhgan Kabiri Chimeh, and Matthew Leach. 2023. FLAME GPU 2: A framework for flexible and performant agent based simulation on GPUs. *Software: Practice and Experience* 53, 8 (2023), 1659–1680.

[19] Dinesh Shenoy and Roberto Rosas. 2018. *Problems & Solutions in Inventory Management*. Springer.

[20] Mirko Stoffers, Daniel Schemmel, Oscar Soria Dustmann, and Klaus Wehrle. 2018. On automated memoization in the field of simulation parameter studies. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28, 4 (2018), 1–25.

[21] Rajan Suri. 1987. Infinitesimal perturbation analysis for general discrete event systems. *Journal of the ACM (JACM)* 34, 3 (1987), 686–717.

[22] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8 (1992), 229–256.