Accelerating ISDA SIMM Margin Optimization with AADC

Application Note | MatLogica | March 2026

Introduction

Under ISDA SIMM, the assignment of trades to counterparties directly determines Initial Margin (IM) requirements. Because SIMM is non-linear — netting offsets within a counterparty, concentration thresholds that cap or amplify risk, and square-root aggregation across buckets — different allocations of the same portfolio can produce materially different margin totals. For a dealer facing multiple counterparties, finding the minimum-margin allocation is a combinatorial optimization problem with significant capital implications.

This note describes a gradient-based approach to SIMM margin optimization and benchmarks three differentiation backends: NumPy finite differences, PyTorch autograd, and AADC adjoint differentiation. All three operate on the same pure-NumPy SIMM engine — only the gradient computation differs.

The Optimization Approach

Continuous Relaxation

The discrete trade-to-dealer assignment is relaxed into a continuous problem:

1. Each trade's hard assignment is replaced by a soft probability vector (softmax over learnable logits), fractionally allocating it across dealers.
2. L-BFGS-B minimizes total SIMM across all dealers, with temperature annealing (τ: $1.0 \rightarrow 0.05$) that progressively sharpens assignments toward discrete.
3. Soft assignments are snapped to hard allocations via argmax, then refined by a greedy local search that tests moving each trade to every other dealer until no improving move exists.

Both phases — gradient optimization and local search — require repeated SIMM evaluations (hundreds to thousands of calls), making per-evaluation runtime the critical bottleneck.

Why Gradients Matter

SIMM is a deeply nested function: weighted sensitivities, intra-bucket correlation, cross-bucket aggregation, concentration risk, cross-risk-class aggregation. Finite differences require $O(n)$ SIMM evaluations per gradient (one per aggregation group per dealer). Adjoint differentiation computes the exact gradient in a single reverse pass — a fundamental algorithmic advantage that grows with problem dimension.

Benchmark Results

Test portfolio: 88 trades, 6 dealers, 583 aggregation groups (2,822 CRIF rows).

Phase	NumPy FD	PyTorch Autograd	AADC Adjoint
Setup (one-time)	—	—	0.2s
Gradient optimization	~26 min	6.0s	0.2s
Local search	~26 min	208s	4.8s
Total	**~26 min**	**214s**	**5.2s**

All three strategies converge to the same optimal allocation (**$3.04B** total SIMM, down from $3.42B before optimization — an **11% reduction**).

Scaling Context

The 88-trade, 6-dealer portfolio is a representative desk-level problem. In production, portfolios of 500+ trades across 10–20 counterparties are common. Because the optimization loop count scales with trades $\times$ dealers, the per-evaluation speedup compounds: a 41$\times$ advantage at 88 trades becomes increasingly decisive at scale, where PyTorch and finite-difference runtimes move from minutes to hours.

How AADC Achieves 41$\times$ Speedup

Compiled Kernel Reuse

AADC records the NumPy SIMM function once into a compiled native kernel. Every subsequent evaluation — forward or adjoint — replays this kernel directly, bypassing Python interpretation entirely. This is especially impactful in local search, where the same SIMM function is called thousands of times with different inputs.

Architecture Mismatch in PyTorch

PyTorch autograd is designed for machine learning workloads: small computational graphs over large tensor operations (matrix multiplies, convolutions). SIMM is the opposite — a large graph of small scalar and vector operations (per-bucket loops, conditional thresholds, piecewise aggregations). PyTorch constructs ~2,928 graph nodes per dealer on every forward pass, then walks them in reverse for gradients. AADC compiles this graph once and evaluates via SIMD-parallel C++, eliminating per-call graph construction overhead.

Zero Code Duplication

A key practical advantage: AADC operates on the **same** `simm_numpy.py` source file used for plain NumPy evaluation. There is no separate AADC-specific engine. The only modifications are mechanical compatibility edits (branchless min/max, safe sqrt at zero) that remain fully backwards-compatible with plain NumPy. PyTorch, by contrast, requires a separate `simm_torch.py` reimplementation — roughly 1,000 lines of duplicated logic that must be kept in sync.

Integration Requirements

Making a NumPy SIMM function AADC-compatible required four categories of mechanical edits, all backwards-compatible:

Square root at zero — `np.sqrt(x)` becomes `np.sqrt(np.maximum(x, 1e-30))`. Avoids infinite adjoint at $x = 0$.

Min/max branching — `np.minimum(a, b)` becomes `(a + b - abs(a - b)) / 2`. Branchless form compatible with AADC active types.

Array construction — `np.empty(n)` becomes `aadc.array(np.empty(n))`. Allows arrays to hold active types; falls back to `np.array` without AADC.

Amount-dependent branches — `if amount > threshold` becomes a structural boolean flag. Branches must be input-independent at record time.

These are one-time edits. Once applied, the same function serves all three backends: plain NumPy, AADC kernel recording, and AADC kernel evaluation.

Conclusion

For iterative, evaluation-heavy problems like margin optimization, the choice of differentiation backend has an outsized impact on total runtime. AADC's record-once, replay-many architecture delivers a **41× end-to-end speedup** over PyTorch autograd on a representative SIMM portfolio — reducing total optimization time from 3.5 minutes to 5 seconds. The same NumPy source code powers both plain evaluation and AADC-accelerated adjoint differentiation, eliminating the maintenance burden of a separate differentiable engine.

As portfolios scale to hundreds of trades and dozens of counterparties, this performance gap widens further, making compiled adjoint differentiation a practical necessity for real-time margin optimization workflows.

AADC (Automatic Adjoint Differentiation Compiler) is developed by MatLogica. For more information, visit matlogica.com.