First, thank all the reviewers for offering valuable suggestions to help to improve the writing of the paper. The reviewers generally agree that our paper is innovative in some aspects, but needs some improvement in the writing. We will keep improving our writing before the camera ready.

Reviewer #1 think our work lack the comparison to strong baselines like TensorFlow and PyTorch. This is not true, Tapenade is a very strong baseline in the field of **generic** AD. TensorFlow and PyTorch are **domain specific** AD software for traditional tensor-based machine learning. Some applications like bundle adjustment in computer vision mainly contains scalar operations. People benchmarked different packages in the bundle adjustment application as shown in the figure, which suggests that we have chosen one of the worlds' best generic AD package as our baseline. Tapenade is 10^{3-5} times faster than PyTorch and TensorFlow. However Tapenade is commercial, close sourced and C/Fortran based. We are proud that NiLang is even better than Tapenade in this benchmark. We don't benchmark the popular Julia package Flux because we benchmarked its backend Zygote instead, where NiLang shows more than one order advantage in time.

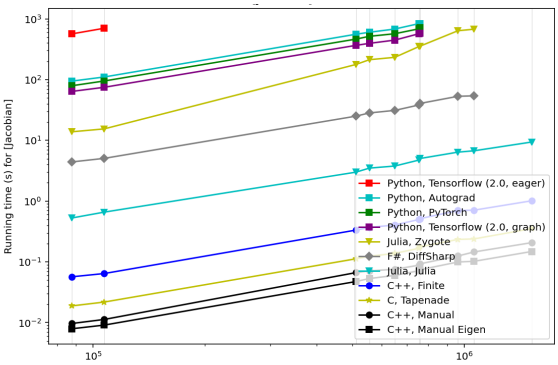

Figure 1: The bundle adjustment benchmark conducted by the ADBench (from the github) project of microsoft [arXiv:1807.10129]. NiLang outperforms all software in this figure.

Reviewer #2 is wondering if reversible computing is equivalent to traditional AD with optimized checkpointing. We agree that with *proper coding style* and the use of reversibility, optimal checkpointing is equivalent to reversible programming. However, the `proper coding style` is very difficult for a user without reversible thinking in mind. In the sparse matrix dot product example in the Appendix. NiLang requires a user to preallocate a `branch_keeper` to store the decisions of branches to enforce reversibility inside the loop. However, if a user is writing it in free style, checkpointing has to insert stack operations inside the loop, which will slow down the program significantly. Not to say the optimal checkpointing itself is a well known hard topic. Allocating automatically for a user is even more dangerous in GPU programming. In NiLang, one can compile the bundle adjustment code to GPU to enjoy a 200x speed up with no more than 10 extra lines of code. It is much easier for users to completely avoid allocation inside a loop in NiLang comparing with checkpointing. Allocation can be the performance killer when differentiating array mutation operations. NiLang supports array mutations naturally. As NiLang is getting more users, some use it to differentiate backtest policies and variational mean field computing. In both cases, NiLang help increases the performance by a factor of $\sim$ 600 comparing with Zygote, just because the array mutations is properly handled. As far as we know, computational graph is not convenient to represent inplace operations. Either the write-once `TensorArray` in TensorFlow or the mutable leaf tensors in PyTorch are not truly mutable. This explains Referee #4's concern that the computational graph can represent any program. Yes, but they are not convenient to describe array mutations.

Reviewer #3 wants to know from which aspect NiLang is different from a traditional reversible programming language, and the limitations of the reversible programming AD. For a long time, the reversible programming languages focus mainly on the theoretical elegance and ignored the productivity. Researchers tried to add functional and object-oriented features to reversible languages. NiLang is special for that it supports many practical elements like arrays, complex numbers, fixed-point numbers, and logarithmic numbers and being an eDSL so that it can be used directly to accelerate machine learning frameworks in Julia. Reversible programming does not have any limitation in describing a computational process, because any program can be written in a reversible form. The most severe weakness of NiLang should be the floating-point arithmetic suffers from the rounding error. We add a seperate section to help people understand the limitations and possible solutions better.

Reviewers #1 and #4 think our result can not be reproduced. Considering both NiLang and the benchmarks are open source on GitHub, we respectfully disagree. Reviewer #1 mention that SVD is available in PyTorch and TensorFlow. It is true. There are more than 10 papers on it and we have cited some of them in the main text. We emphasis that they are **manually** generated. We say, "why bother 10 papers? just let the generic AD do its job!"

At last, we encourage reviewers to view this project more from the "future" perspective. Nowadays, energy is becoming one of the most deadly bottlenecks for machine learning applications. From a physicist's perspective, we believe that reversible computing is the only correct approach to solve energy conundrums. Classical reversible computing has been silenced for $\sim$15 years, NiLang is trying to bridge the new trend machine learning and reversible computing. We will try our best to convey this point better in the updated manuscript. As Reviewer #2 said, it is a long overdue.