Differentiate Everything with a Reversible Domain-Specific Language

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Traditional machine reverse-mode automatic differentiation (AD) suffers from the problem of having a space overhead that linear to time in order to trace back the computational state, which is also the source of poor performance. In reversible programming, a program can be executed bi-directionally, which means we do not need any additional design to trace back the computational state. This paper answers how practical it is to implement a programming language level reverse mode AD in a reversible programming language. By implementing sparse matrix operations and some machine learning applications in our reversible eDSL NiLang, and benchmark the performance with state-of-the-art AD frameworks, our answer is a definite positive. NiLang is an open-source r-Turing complete reversible eDSL in Julia. It empowers users the flexibility to tradeoff time, space, and energy rather than caching data into a global tape. Manageable memory allocation makes it an excellent tool to differentiate GPU kernels too.

1 Introduction

Most popular autmatic differetiation (AD) packages in the market, such as TensorFlow Abadi *et al.* (2015), Pytorch Paszke *et al.* (2017) and Flux Innes *et al.* (2018) implements reverse mode AD at the tensor level to meet the need of machine learning. These frameworks sometimes fail to meet the diverse needs in research, for example, in physics research,

1. People need to differentiate over sparse matrix operations that are important for Hamiltonian engineering Hao Xie and Wang, like solving dominant eigenvalues and eigenvectors Golub and Van Loan (2012),
2. People need to backpropagate singular value decomposition (SVD) function and QR decomposition in tensor network algorithms to study the phase transition problem Golub and Van Loan (2012); Liao *et al.* (2019); Seeger *et al.* (2017); Wan and Zhang (2019); Hubig (2019),
3. People need to differentiate over a quantum simulation where each quantum gate is an inplace function that changes the quantum register directly Luo *et al.* (2019).

People have to keep adding new backward rules to the function pool. In the remaining text, we call this type of AD the domain specific AD (DS-AD). To meet the diversed need, we need a general purposed AD (GP-AD) too to differentiate the whole programming language. Some source code transformation based AD packages like Tapenade Hascoet and Pascual (2013) and Zygote Innes (2018); Innes *et al.* (2019) are close to this goal. They read the source code from a user and generate a new code that computes the gradients. However, these packages have their own limitations too. In many practical applications, differentiating a program might do billions of computations. Frequent caching of data slows down the program significantly, and the memory usage will become a bottleneck as well. It is still not practical to automatically generate the backward rules for BLAS functions and sparse matrix operations with a performance comparable to the state-of-the-art.

We propose to implement GP-AD on a reversible (domain-specific) programming language Perumalla (2013); Frank (2017). So that the intermediate states of a program can be traced backward with no extra effort. There have been many prototypes of reversible languages like Janus Lutz (1986), R (not the popular one) Frank (1997), Erlang Lanese *et al.* (2018) and object-oriented ROOPL Haulund (2017). In the past, the primary motivation of studying reversible programming is to support reversible computing devices Frank and Knight Jr (1999) like adiabatic complementary metal–oxide–semiconductor (CMOS) Koller and Athas (1992), molecular mechanical computing system Merkle *et al.* (2018) and superconducting system Likharev (1977); Semenov *et al.* (2003), where a reversible computing device is more energy-efficient from the perspective of information and entropy, or by the Landauer's principle Landauer (1961). However, the field of reversible computing faces the issue of having not enough funding in recent decade Frank (2017). As a result, not many people in machine learning are familiar with some marvelous designs in reversible computing. People have not connected it with automatic differentiation seriously, even though they share so many similarities.

In machine learning, people also manage to not erasing informations that needed to . In many neural networks, people also use reversibility to reduce the memory cost. However, we find when when people are trying to write a program in a free style, it is generally hard for an AD engine to avoid allocations. Especially when the allocation is inside the loop, it will become a performance killer. What's worse, allocating variables in a global stack will make the code impossible to be compiled to GPU even if it is generically written.

This paper aims to break the information barrier between the machine learning community and the reversible programming community and provide yet another strong motivation to develop reversible programming. In this paper, we first introduce the language design of a reversible programming language and introduce our reversible eDSL NiLang in Sec. 2. In Sec. 3, we explain the implementation of automatic differentiation in this eDSL. In Sec. 4, we benchmark the performance of NiLang with other AD packages and explain why reversible programming AD is fast. In the appendix, we show the detailed language design of NiLang, show some examples used in the benchmark, discuss several important issues including the time-space tradeoff, reversible instructions and hardware, and finally, an outlook to some open problems to be solved.

2 Language design

2.1 Introductions to reversible language design

In a modern programming language, functions are pushed to a global stack for scheduling. The memory layout of a function consists of input arguments, a function frame with information like the return address and saved memory segments, local variables, and working stack. After the call, the function clears run-time information, only stores the return value. In reversible programming, this kind of design is no longer the best practice. One can not discard input variables and local variables easily after a function call, since discarding information may ruin reversibility. For this reason, reversible functions are very different from irreversible ones from multiple perspectives.

2.1.1 Memory management

A distinct feature of reversible memory management is, the content of a variable must be known when it is deallocated. We denote the allocation of a zero emptied memory as $\texttt{x} \leftarrow \mathbb{0}$, and the corresponding deallocation as $\texttt{x} \rightarrow \mathbb{0}$. A variable x can be allocated and deallocated in a local scope, which is called an ancilla. It can also be pushed to a stack and used later with a pop statement. This stack is similar to a traditional stack, except it zero-clears the variable after pushing and presupposes that the variable being zero-cleared before popping. Knowing the contents in the memory when deallocating is not easy. Hence Charles H. Bennett introduced the famous compute-copy-uncompute paradigm Bennett (1973) for reversible programming.

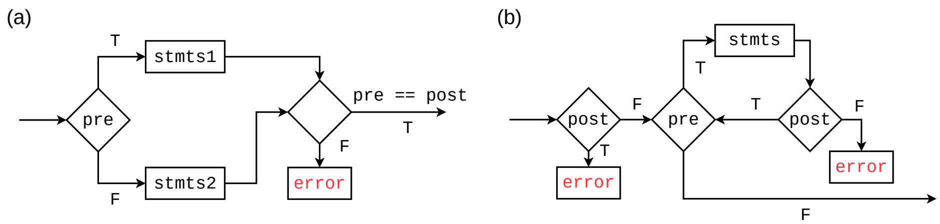

Figure 1: The flow chart for reversible (a) `if` statement and (b) `while` statement. "pre" and "post" represents precondition and postconditions respectively.

2.1.2 Control flows

One can define reversible `if`, `for` and `while` statements in a slightly different way comparing with its irreversible counterpart. The reversible `if` statement is shown in Fig. 1 (a). Its condition statement contains two parts, a precondition and a postcondition. The precondition decides which branch to enter in the forward execution, while the postcondition decides which branch to enter in the backward execution. After executing the specific branch, the program checks the consistency between precondition and postcondition to make sure they are consistent. The reversible `while` statement in Fig. 1 (b) also has two condition fields. Before executing the condition expressions, the program preassumes the postcondition is false. After each iteration, the program asserts the postcondition to be true. In the reverse pass, we exchange the precondition and postcondition. The reversible `for` statement is similar to irreversible ones except that after executing the loop, the program checks the values of these variables to make sure they are not changed. In the reverse pass, we exchange `start` and `stop` and inverse the sign of `step`.

2.1.3 Arithmetic instructions

Every arithmetic instruction has a unique inverse that can undo the changes.

- For logical operations, `y ⊻= f(args...)` is self reversible.
- For integer and floating point arithmetic operations, we treat `y += f(args...)` and `y -= f(args...)` as reversible to each other. Here `f` can be an arbitrary pure function such as `identity`, `*`, `/` and `^`. Let's forget the floating point rounding errors for the moment and discuss in detail in the supplimentary materials.
- For logartihmic number and tropical number algebra Speyer and Sturmfels (2009), `y *= f(args...)` and `y /= f(args...)` as reversible to each other. Notice the zero element ($-\infty$) in the Tropical algebra is not considered here.

Besides the above two types of operations, `SWAP` operation that exchanges the contents in two memory spaces is also widely used in reversible computing systems.

2.2 Differetiable Reversible eDSL: NiLang

We develop an embedded domain-specific language (eDSL) NiLang in Julia language Bezanson *et al.* (2012, 2017) that implements reversible programming. One can write reversible control flows, instructions, and memory managements inside this macro. Julia is a popular language for scientific programming. We choose Julia as the host language for multiple purposes. The most important consideration is speed that crucial for a GP-AD. Its clever design of type inference and just in time compiling provides a C like speed. Also, it has a rich ecosystem for meta-programming. The package for pattern matching MLStyle allow us to define an eDSL conveniently. Last but not least, its multiple-dispatch provides the polymorphism that will be used in our AD engine. The main feature of NiLang is contained in a single macro `@i` that compiles a reversible function. We can use `macroexpand` to show the compiling a reversible function to the native Julia function.

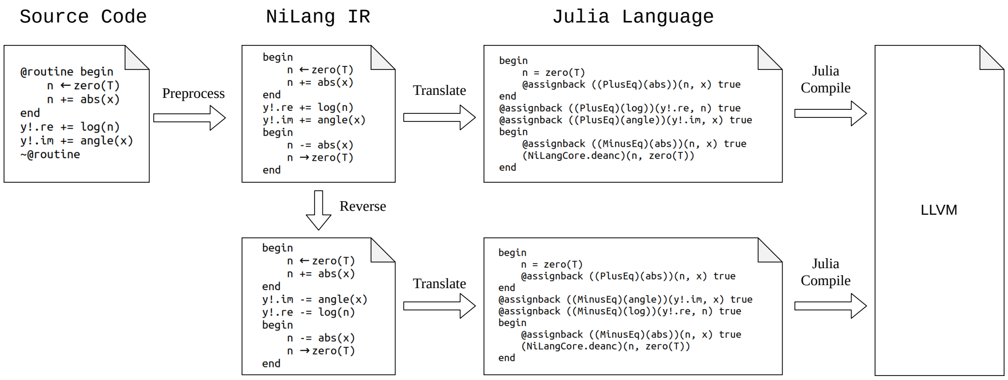

Figure 2: Compiling the body of the complex valued log function defined in Listing. 1.

```julia
julia> using NiLangCore, MacroTools

julia> MacroTools.prettify(@macroexpand @i function f(x, y)
           SWAP(x, y)
       end)
quote
    $(Expr(:meta, :doc))
    function $(Expr(:where, :(f(x, y))))
        dove = wrap_tuple(SWAP(x, y))
        x = dove[1]
        y = dove[2]
        (x, y)
    end
    if NiLangCore._typeof(f) != _typeof(~f)
        function $(Expr(:where, :((hummingbird::_typeof(~f))(x, y))))
            toad = wrap_tuple((~SWAP)(x, y))
            x = toad[1]
            y = toad[2]
            (x, y)
        end
    end
end
```

Here, the version of NiLang is v0.4.0. Macro `@i` generates two functions that reversible to each other `f` and `~f`. `~f` is an callable of type `Inv{typeof(f)}`, where the type parameter `typeof(f)` stands for the type of the function `f`. In the body of `f`, `NiLangCore.wrap_tuple` is used to unify output data types to tuples. The outputs of `SWAP` are assigned back to its input variables. At the end of this function, this macro attaches a return statement that returns all input variables.

The compilation of a reversible function to native Julia functions is consisted of three stages: *preprocessing*, *reversing* and *translation*. Fig. 2 shows the compilation of the complex valued log function body, which is originally defined as follows.

Listing 1: Reversible complex valued log function $y \mathrel{+}= \log(|x|) + i\mathrm{Arg}(x)$.

```julia
@i function (:+=)(log)(y!::Complex{T}, x::Complex{T}) where T
    @routine begin
        n ← zero(T)
        n += abs(x)
    end
    y!.re += log(n)
    y!.im += angle(x)
    ~@routine
end
```

In the *preprocessing* stage, the compiler pre-processes human inputs to reversible NiLang IR. The preprocessor removes redundant grammars and expands shortcuts. In the left most code box in Fig. 2, one uses `@routine <stmt>` statement to record a statement, and `~@routine` to insert the

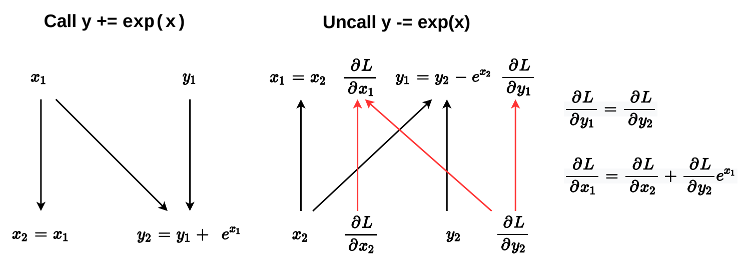

Figure 3: Binding the adjoint rule of `y+=exp(x)` to its uncomputing program.

corresponding inverse statement for uncomputing. The computing-uncomputing macros `@routine` and `~@routine` is expanded in this stage. Here, one can input "←" and "→" by typing "\leftarrow[TAB KEY]" and "\rightarrow[TAB KEY]" respectively in a Julia editor or REPL. In the *reversing* stage, based on this symmetric and reversible IR, the compiler generates reversed statements. In the *translation* stage, the compiler translates this reversible IR as well as its inverse to native Julia code. It adds `@assignback` before each function call, inserts codes for reversibility check, and handle control flows. We can expand the `@assignback` macro to see the compiled expression. As a final step, the compiler attaches a return statement that returns all updated input arguments at the end of a function definition. Now, the function is ready to execute on the host language.

3 Reversible automatic differentiation

3.1 First order gradient

The computation of gradients in NiLang contains two parts, computing and uncomputing. In the computing stage, the program marches forward and computes outputs. In the uncomputing stage, we attach each scalar and array element with an extra gradient field and feed them into the inverse function. To composite data type with a gradient field is called `GVar`. As shown in Fig. 3, when an instruction is uncalled, we first uncompute the value field of `GVar`s to x_1 and y_1, using the input information, we then update the gradient fields according to the formula in the right panel. The binding utilizes the multiple dispatch in Julia, where a function can be dynamically dispatched based on the run time type of more than one of its arguments. Here, we dispatch a inverse instruction with input type `GVar` to the `(:-=)(exp)` instruction.

```
@i @inline function (:-=)(exp)(out!::GVar, x::GVar{T}) where T
    @routine @invcheckoff begin
        anc1 ← zero(value(x))
        anc1 += exp(value(x))
    end
    value(out!) -= identity(anc1)
    grad(x) += grad(out!) * anc1
    ~@routine
end
```

Here, the first four lines is the `@routine` statement that computes e^{x_2} and store the value into an ancilla. The 5th line updates the `value` dataview of `out!`. The 6th line updates the gradient fields of `x` and `y` by applying the adjoint rule of `(:+=)(exp)`. Finally, `@routine` uncomputes `anc1` so that it can be returned to the "memory pool". One does not need to define a similar function on `(:+=)(exp)` because macro `@i` will generate it automatically. Notice that taking inverse and computing gradients commute McInerney (2015).

3.2 Hessians

Combining the uncomputing program in NiLang with dual-numbers is a simple yet efficient way to obtain Hessians. The dual number is the scalar type for computing gradients in the forward mode AD, it wraps the original scalar with a extra gradient field. The gradient field of a dual number is updated automatically as the computation marches forward. By wrapping the elementary type with `Dual` defined in package ForwardDiff Revels *et al.* (2016) and throwing it into the gradient program defined in NiLang, one obtains one row/column of the Hessian matrix straightforward. We will show a benchmark in Sec. 4.2.

3.3 Complex numbers

To differentiate complex numbers, we re-implemented complex instructions reversibly. For example, with the reversible function defined in in Listing. 1, we can differentiated complex valued log with no extra effort.

3.4 CUDA kernels

CUDA programming is playing a significant role in high-performance computing. In Julia, one can write GPU compatible functions in native Julia language with KernelAbstractions Besard *et al.* (2017). Since NiLang does not push variables into stack automatically for users, it is safe to write differentiable GPU kernels with NiLang. We will show this feature in the benchmarks of bundle adjustment (BA) in Sec. 4.3. Here, one should notice that the shared read in forward pass will become shared write in the backward pass, which may result in incorrect gradients. We will review this issue in the supplimentary material.

4 Benchmarks

In the following benchmarks, the CPU device is Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz, and the GPU device is Nvidia Titan V. For NiLang benchmarks, we have turned off the reversibility check off to achieve a better performance. Codes used in benchmarks could be found the in Examples section of the supplimentary material.

4.1 Sparse matrices

We benchmarked the call, uncall and backward propagation time used for sparse matrix dot product and matrix multiplication. Here, we estimate the time for back propagating gradients rather than including both forward and backward, since `mul!` does not output a scalar as loss.

	`dot`	`mul!` (complex valued)
Julia-O	3.493e-04	8.005e-05
NiLang-O	4.675e-04	9.332e-05
NiLang-B	5.821e-04	2.214e-04

Table 1: Absolute runtimes in seconds for computing the objectives (O) and the backward pass (B) of sparse matrix operations. The matrix size is 1000×1000, and the element density is 0.05. The total time used in computing gradient can be estimated by summing "O" and "B".

The time used for computing backward pass is approximately 1.5-3 times the Julia's native forward pass. This is because the instruction length of differentiating basic arithmetic instructions is longer than pure computing.

4.2 Graph embedding problem

Since one can combine ForwardDiff and NiLang to obtain Hessians, it is interesting to see how much performance we can get in differentiating the graph embedding program. The problem definition could be found in the supplimentary material.

In Table 2, we show the the performance of different implementations by varying the dimension k. The number of parameters is $10k$. As the baseline, (a) shows the time for computing the function call. We have reversible and irreversible implementations, where the reversible program is slower

k	2	4	6	8	10
Julia-O	4.477e-06	4.729e-06	4.959e-06	5.196e-06	5.567e-06
NiLang-O	7.173e-06	7.783e-06	8.558e-06	9.212e-06	1.002e-05
NiLang-U	7.453e-06	7.839e-06	8.464e-06	9.298e-06	1.054e-05
NiLang-G	1.509e-05	1.690e-05	1.872e-05	2.076e-05	2.266e-05
ReverseDiff-G	2.823e-05	4.582e-05	6.045e-05	7.651e-05	9.666e-05
ForwardDiff-G	1.518e-05	4.053e-05	6.732e-05	1.184e-04	1.701e-04
Zygote-G	5.315e-04	5.570e-04	5.811e-04	6.096e-04	6.396e-04
(NiLang+F)-H	4.528e-04	1.025e-03	1.740e-03	2.577e-03	3.558e-03
ForwardDiff-H	2.378e-04	2.380e-03	6.903e-03	1.967e-02	3.978e-02
(ReverseDiff+F)-H	1.966e-03	6.058e-03	1.225e-02	2.035e-02	3.140e-02

Table 2: Absolute times in seconds for computing the objectives (O), uncall objective (U), gradients (G) and Hessians (H) of the graph embedding program. k is the embedding dimension, the number of parameters is $10k$.

than the irreversible native Julia program by a factor of ~ 2 due to the uncomputing overhead. The reversible program shows the advantage of obtaining gradients when the dimension $k \geq 3$. The larger the number of inputs, the more advantage it shows due to the overhead proportional to input size in forward mode AD. The same reason applies to computing Hessians, where the combo of NiLang and ForwardDiff gives the best performance for $k \geq 3$.

4.3 Gaussian mixture model and bundle adjustment

We reproduced the benchmarks for Gaussian mixture model (GMM) and bundle adjustment (BA) in Srajer *et al.* (2018) by re-writing the programs in a reversible style. We show the results in Table 3 and Table 4. In our new benchmarks, we also rewrite the ForwardDiff program for a fair benchmark, this explains the difference between our results and the original benchmark. The Tapenade data is obtained by executing the docker file provided by the original benchmark, which provides a baseline for comparison.

# parameters	3.00e+1	3.30e+2	1.20e+3	3.30e+3	1.07e+4	2.15e+4	5.36e+4	4.29e+5
Julia-O	9.844e-03	1.166e-02	2.797e-01	9.745e-02	3.903e-02	7.476e-02	2.284e-01	3.593e+00
NiLang-O	3.655e-03	1.425e-02	1.040e-01	1.389e-01	7.388e-02	1.491e-01	4.176e-01	5.462e+00
Tapende-O	1.484e-03	3.747e-03	4.836e-02	3.578e-02	5.314e-02	1.069e-01	2.583e-01	2.200e+00
ForwardDiff-G	3.551e-02	1.673e+00	4.811e+01	1.599e+02	-	-	-	-
NiLang-G	9.102e-03	3.709e-02	2.830e-01	3.556e-01	6.652e-01	1.449e+00	3.590e+00	3.342e+01
Tapenade-G	5.484e-03	1.434e-02	2.205e-01	1.497e-01	4.396e-01	9.588e-01	2.586e+00	2.442e+01

Table 3: Absolute runtimes in seconds for computing the objective (O) and gradients (G) of GMM with 10k data points. "-" represents missing data due to not finishing the computing in limited time.

In the GMM benchmark, NiLang's objective function has overhead comparing with irreversible programs in most cases. Except the uncomputing overhead, it is also because our naive reversible matrix-vector multiplication is much slower than the highly optimized BLAS function, where the matrix-vector multiplication is the bottleneck of the computation. The forward mode AD suffers from too large input dimension in the large number of parameters regime. Although ForwardDiff batches the gradient fields, the overhead proportional to input size still dominates. The source to source AD framework Tapenade is faster than NiLang in all scales of input parameters, but the ratio between computing the gradients and the objective function are close.

In the BA benchmark, reverse mode AD shows slight advantage over ForwardDiff. The bottleneck of computing this large sparse Jacobian is computing the Jacobian of a elementary function with 15 input arguments and 2 output arguments, where input space is larger than the output space. In this instance, our reversible implementation is even faster than the source code transformation based AD framework Tapenade. With KernelAbstractions, we run our zero allocation reversible program on GPU, which provides a >200x speed up.

# measurements	3.18e+4	2.04e+5	2.87e+5	5.64e+5	1.09e+6	4.75e+6	9.13e+6
Julia-O	2.020e-03	1.292e-02	1.812e-02	3.563e-02	6.904e-02	3.447e-01	6.671e-01
NiLang-O	2.708e-03	1.757e-02	2.438e-02	4.877e-02	9.536e-02	4.170e-01	8.020e-01
Tapenade-O	1.632e-03	1.056e-02	1.540e-02	2.927e-02	5.687e-02	2.481e-01	4.780e-01
ForwardDiff-J	6.579e-02	5.342e-01	7.369e-01	1.469e+00	2.878e+00	1.294e+01	2.648e+01
NiLang-J	1.651e-02	1.182e-01	1.668e-01	3.273e-01	6.375e-01	2.785e+00	5.535e+00
NiLang-J (GPU)	1.354e-04	4.329e-04	5.997e-04	1.735e-03	2.861e-03	1.021e-02	2.179e-02
Tapenade-J	1.940e-02	1.255e-01	1.769e-01	3.489e-01	6.720e-01	2.935e+00	6.027e+00

Table 4: Absolute runtimes in seconds for computing the objective (O) and Jacobians (J) in bundle adjustment.

Broader Impact

Our automatic differentiation in a reversible eDSL brings the field of reversible computing to the modern context. We believe it will be accepted by the public to meet current scientific automatic differentiation needs and aim for future energy-efficient reversible devices. For solving practical issues, in an unpublished paper, we have successfully differentiated a spin-glass solver to find the optimal configuration on a 28×28 square lattice in a reasonable time. There are also some interesting applications like normalizing flow and bundle adjustment in the example folder of NiLang repository and JuliaReverse organization. For the future, energy consumption is an even more fundamental issue than computing time and memory. Current computing devices, including CPU, GPU, TPU, and NPU consume much energy, which will finally hit the "energy wall". We must get prepared for the technical evolution of reversible computing (quantum or classical), which may cost several orders less energy than current devices.

We also see some drawbacks to the current design. It requires the programmer to change to programing style rather than put effort into optimizing regular codes. It is not fully compatible with modern software stacks. Everything, including instruction sets and BLAS functions, should be redesigned to support reversible programming better. We put more potential issues and opportunities in the discussion section of the supplementary material. Solving these issues requires the participation of people from multiple fields.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," (2015), software available from tensorflow.org.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, in *NIPS Autodiff Workshop* (2017).

M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah, "Fashionable modelling with flux," (2018), arXiv:1811.01457 [cs.PL] .

J.-G. L. Hao Xie and L. Wang, arXiv:2001.04121 .

G. H. Golub and C. F. Van Loan, *Matrix computations*, Vol. 3 (JHU press, 2012).

H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, Physical Review X **9** (2019), 10.1103/physrevx.9.031041.

M. Seeger, A. Hetzel, Z. Dai, E. Meissner, and N. D. Lawrence, "Auto-differentiating linear algebra," (2017), arXiv:1710.08717 [cs.MS] .

Z.-Q. Wan and S.-X. Zhang, "Automatic differentiation for complex valued svd," (2019), arXiv:1909.02659 [math.NA] .

C. Hubig, "Use and implementation of autodifferentiation in tensor network methods with complex scalars," (2019), arXiv:1907.13422 [cond-mat.str-el] .

X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, "Yao.jl: Extensible, efficient framework for quantum algorithm design," (2019), arXiv:1912.10877 [quant-ph] .

L. Hascoet and V. Pascual, ACM Transactions on Mathematical Software (TOMS) **39**, 20 (2013).

M. Innes, "Don't unroll adjoint: Differentiating ssa-form programs," (2018), arXiv:1810.07951 [cs.PL] .

M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, CoRR **abs/1907.07587** (2019), arXiv:1907.07587 .

K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).

M. P. Frank, IEEE Spectrum **54**, 32–37 (2017).

C. Lutz, "Janus: a time-reversible language," (1986), *Letter to R. Landauer*.

M. P. Frank, *The R programming language and compiler*, Tech. Rep. (MIT Reversible Computing Project Memo, 1997).

I. Lanese, N. Nishida, A. Palacios, and G. Vidal, Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018).

T. Haulund, "Design and implementation of a reversible object-oriented programming language," (2017), arXiv:1707.07845 [cs.PL] .

M. P. Frank and T. F. Knight Jr, *Reversibility for efficient computing*, Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and … (1999).

J. G. Koller and W. C. Athas, in *Workshop on Physics and Computation* (1992) pp. 267–270.

R. C. Merkle, R. A. Freitas, T. Hogg, T. E. Moore, M. S. Moses, and J. Ryley, Journal of Mechanisms and Robotics **10** (2018), 10.1115/1.4041209.

K. Likharev, IEEE Transactions on Magnetics **13**, 242 (1977).

V. K. Semenov, G. V. Danilov, and D. V. Averin, IEEE Transactions on Applied Superconductivity **13**, 938 (2003).

R. Landauer, IBM journal of research and development **5**, 183 (1961).

C. H. Bennett (1973).

D. Speyer and B. Sturmfels, Mathematics Magazine **82**, 163 (2009).

J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, arXiv preprint arXiv:1209.5145 (2012).

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM Review **59**, 65–98 (2017).

A. McInerney, *First steps in differential geometry* (Springer, 2015).

J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," (2016), arXiv:1607.07892 [cs.MS] .

T. Besard, C. Foket, and B. D. Sutter, CoRR **abs/1712.03112** (2017), arXiv:1712.03112 .

F. Srajer, Z. Kukelova, and A. Fitzgibbon, Optimization Methods and Software **33**, 889 (2018).