Instruction level automatic differentiation on a reversible Turing machine

Jin-Guo Liu,[1,*] Taine Zhao,[2] and [JG: MLS's name here][3]

[1]*Institute of Physics, Chinese Academy of Sciences, Beijing 100190, China*
[2]*Department of Computer Science, University of Tsukuba*
[3]*MLS's affiliations here*

This paper considers source to source automatic differentiation (AD) on a reversible Turing machine. We start by reviewing why adjoint mode AD is hard for traditional machine learning frameworks. Then we propose a solution to existing issues by writing a program reversiblly. We developed an reversible eDSL NiLang in Julia that can differentiate a general program while being compatible with Julia ecosystem. It can generate efficient backward rules with reasonable time and memory efficiency. We demonstrate its power by generating the backard rules for Bessel function, sparse matrix dot product, unitary matrix multiplication and QR decomposition. It can also be used to solve a graph embeding problem efficiently. Also, we will discuss the challenges that we face towards rigorous reversible programming from the instruction and hardware perspective.

I. INTRODUCTION

Computing the gradients of a numeric model $f : \mathbb{R}^m \to \mathbb{R}^n$ plays a crucial role in scientific computing. Consider a computing process

$$\begin{aligned}\mathbf{x}^1 &= f_1(\mathbf{x}^0)\\ \mathbf{x}^2 &= f_2(\mathbf{x}^1)\\ &\cdots\\ \mathbf{x}^L &= f_L(\mathbf{x}^{L-1})\end{aligned}$$

where $x^0 \in R^m$, $x^L \in R^n$, L is the depth of computing. The Jacobian of this program is a $n \times m$ matrix $J_{ij} \equiv \frac{\partial x_i^L}{\partial x_j^0}$, where x_j^0 and x_i^L are single elements of inputs and outputs. Computing part of the Jacobian automatically is what we called automatic differentiation (AD). It can be classified into three classes, the tangent mode AD, the adjoint mode AD and the mixed mode AD. [1] The tangent mode AD computes the Jacobian matrix elements that related to a single input using the chain rule $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}} \frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$, while a tangent mode AD computes Jacobian matrix elements that related to a single output using $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}} \frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$. In variational applications where the loss function always outputs a scalar, the adjoint mode AD is preferred. However, implementing adjoint mode AD is harder than implementing its tangent mode counterpart, because it requires propagating the gradients in the inverse direction of computing the loss. The back propagation of gradients requires intermediate information of a program that includes

1. the computational process,
2. and variables used in computing gradients.

The computational process is often stored in a computational graph, which is a directed acyclic graph (DAG) that represents the relationship between data and functions. There are two basic techniques for implementation of computational graph, operator overloading and source code transformation. Most popular AD implementations in the market are based on operator overloading. These packages provide a finite set of primitive functions with predefined backward rules. In Pytorch [2] and Flux [3], every variable has a tracker field. When applying a predefined primitive function on a variable, the variable's tracker field keeps track of this function as well as data that needed in later back propagation. TensorFlow [4] uses the similar approach expect it builds a static computational graph as a description of the program before actual computation happens. In research, people need new primitives every now and then, these packages can not cover all the diversed needs in different fields. Then it relies on users to code backward rules manually. For example, in physics, the requirements to AD are quite diversed.

1. We need to differentiate over sparse matrix operations that are important for Hamiltonian engineering [5], like solving dominant eigenvalues and eigenvectors [6].
2. We need to back propagating singular value decomposition (SVD) function and QR decomposition in tensor network algorithms to study the phase transition problem [6, 7].
3. We need to differentiate over a quantum simulation where each quantum gate is a inplace function that changes the quantum register directly [8].

None of the above packages can meet all these requirements by using predefined primitives only. Scientists put lots of effort into deriving backward rules. In the back propagation of dominant eigensolver [5], people managed to circumvent the sparse matrix issue by allowing users to provide the backward function for sparse matrix generator. The backward rules for SVD and eigensolvers has been manually derived recent years [9–11]. One can obtain the gradients correctly for both real numbers and complex numbers [10] in most cases, expect when the spectrum is degenerate. In variational quantum simulator Yao [8], authors implemented a builtin cache free AD engine by utilizing the reversible nature of quantum computing. They derive and implement the backward rule for each type of quantum gate.

Souce code transformation based AD brings hope to free scientists from deriving backard rules. Tools like Tapenade [1]

* cacate0129@iphy.ac.cn

and Zygote [12, 13] generate the adjoint code statically while putting variable on a stack called the wengert list. However, this approach has its own problem too. In traditional AD applications[**JG: what is traditional AD applications**], a program that might do billions of computations will as well get a wengert list in the range of GBs. Frequent caching of data slows down the program significantly, the memory will become a bottleneck as well. In both machine learning and scientific computation, the memory management in AD is becoming a wall [8] that limiting the scale of many applications. In many deep learning models like recurrent neural network [14] and residual neural networks [15], the depth can reach several thousand, where the memory is often the bottleneck of these programs. The memory wall problem is even more sevie when one runs the application on GPU. The computational power of a Nvidia V100 GPU can reaches 100 TFLOPS, competing a small cluster. However, its memory is only 32GB. Back propagating a general program using source code transformation just makes the case worse, because the memory consumption of the program is typically $O(T)$ where T is the runtime of the program. It is nearly impossible to automatically generate the backward rule for SVD with reasonable performance. A better solution to memory management must be found to make source to source AD practical.

We tackle this problem by writting a program reversibly. Reversibility has been used in reduce the memory allocations in machine learning models such as recurrent neural networks [16], Hyperparameter learning [17] and residual neural networks [18], where information buffer [17] and reversible activation functions [19, 20] are used to decrease the memory usage. Our approach makes reversibility a language feature hence is more general purposed. We develop an embedded domain-specific language (eDSL) NiLang in Julia language [21, 22] that implements reversible programming. [23, 24]. This eDSL provides a macro to generate reversible functions that can be used by other programs. One can write reversible control flows, instructions and memory managements inside this macro. We choose Julia as the host language for multiple purposes. Julia is a popular language for scientific programming. Its meta programming and its package for pattern matching MLStyle [25] allow us to define an eDSL conveniently. Its type inference and just in time compiling can remove most overheads introduced in our eDSL, providing a reasonable performance. Most importantly, its multiple dispatch provides the polymorphism that will be used in our autodiff engine. In the past, we see a lot of implementations of AD languages work as an indepedant language. However, we wish our eDSL can benefit people in Julia community directly.

There haven't been any reversible eDSL in Julia before, but there have been many prototypes of reversible languages like Janus [26], R (not the popular one) [27], Erlang [28] and object-oriented ROOPL [29]. In the past, the primary motivation of studying reversible programming is to support reversible devices [30] like adiabatic complementary metal–oxide–semiconductor (CMOS) [31], molecular mechanical computing system [32] and superconducting system [33, 34]. Reversible computing are more energy efficient from information and entropy's perspective, or by the Landauer's principle [35]. After decades of efforts, reversible computing devices are very close to providing productivity now. As an exmaple, adiabatic CMOS can be a better choice already in a spacecraft [36], where energy is more valuable than device itself. Reversible programming are interesting to software engineers too, because it is a powerful tool to schedule asynchronious events [37] and debug a program bidirectionally [38]. However, the field of reversible computing faces the difficulty of not enough funding recent decade [24]. As a result, not much people studying AD know the marvelous designs in reversible computing. People haven't connected it with automatic differentiation seriously even though they are so related. We aims to breaks the information barrier between the machine learning community and the reversible programming community in our work, and provides yet another strong motivation to develop reversible programming.

In this paper, we first introduce the language design of NiLang in Sec. II. In Sec. III, we explain the back-propagation algorithm of Jacobians and Hessians in this eDSL. In Sec. IV, we show several examples including Bessel function, dot product between sparse matrices, unitary matrix multiplication and QR decomposition and reversible CUDA kernel for quantum simulation. We show how to generate first order and second order backward rules for these functions. In Sec. V, we discuss several important issues, how time-space tradeoff works, reversible instructions and hardware, and finally, an outlook to some open problems to be solved. In the appendix, we show the grammar of NiLang and other technical details.

II. LANGUAGE DESIGN

A. Introductions to reversible language design

[**JG: how about now? feel free to comment more**] [**MLS: Add more introduction to reversible programming**] 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 these run-time information, only stores the return value. In the 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.

1. *Memory management*

The key difference between reversible and irreversible memory management is the content of a variable that going to be discarded in a reversible program must be known. We denote the allocation of a zero emptied memory as $\texttt{x} \leftarrow \texttt{0}$, and the corresponding deallocation as $\texttt{x} \rightarrow \texttt{0}$. A variable x

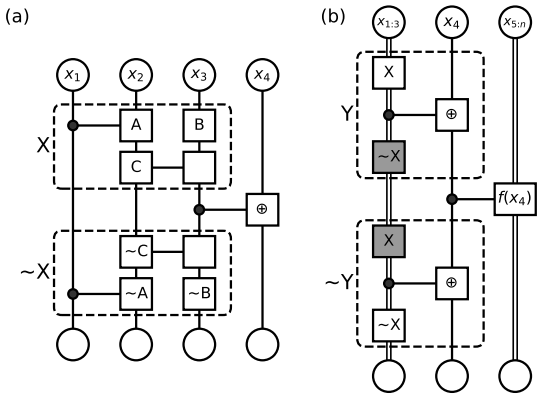


Figure 1: Data oriented computational graph that shows the compute-copy-uncompute paradigm. Circles connected by vertical lines are variables, where double lines represents multiple variables. Boxes and dots connected by parallel lines are operations. A dot on a line represents a control parameter while a box represents a mutable parameter.

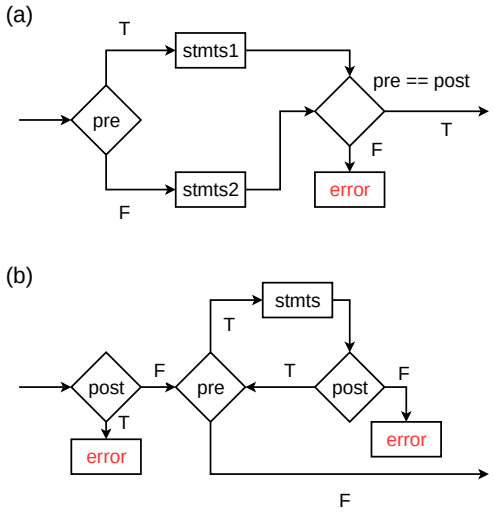


Figure 2: Flow chart for reversible (a) `if` statement and (b) `while` statement. "stmts", "stmts1" and "stmts2" are statements, statements in true branch and statements in false branch respectively. "pre" and "post" are precondition and postconditions respectively.

can be allocated and deallocated in a local scope, which is called an ancilla. It can also be push 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 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 [39]. In order to show how reversible memory manage works, we introduce the memory oriented computational graph as shown in Fig. 1. Notations are highly inspired by quantum circuit representations. A vertical line is a variable, it can be used by multiple operations. Hence it is a hypergraph rather than a simple graph like DAG. When a variable is used by a function, depending on whether its value is changed by this function or not, we put a box or a dot at the cross respectively. Let's consider the example program shown in panel (a). The subrogram in dashed box `X` is executed on space $x_{1:3}$ to compute desired result, which we call the computing stage. In the copying stage, the contents in x_3 is read out to a pre-emptied memory x_4 through addition operation $\oplus$, this is the piece of information that we care. Since this copy operation does not change contents of $x_{1:3}$, we can use the inverse operation ~`X` to undo all the changes to these registers. If a variable in $x_{1:3}$ is initialized as a known value like 0, now it can be deallocated since its value is known again. If this subroutine of generating x_4 is used in another program as shown in Fig. 1 (b), x_4 can be uncomputed by the reversing the whole subroutine in panel (a). The interesting fact is, both X and $\sim X$ are executed twice in this program, which looks like unnecessary. We can of cause cancel a pair of X and $\sim X$ (the gray boxes), by doing this, we are not allowed to deallocate the memory $x_{1:3}$ during computing $f(x_4)$, i.e. additional space is required. The trade off between space and time will be discussed in detail in Sec. V A.

2. *Control flows*

The reversible `if` statement is shown in Fig. 2 (a). It contains two conditions, 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 the consistent. The reversible `while` statement is shown in Fig. 2 (b). It also has both precondition and postcondition. 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 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.

3. *Arithmetic instructions*

Every arithmetic instruction has a unique inverse that can undo the changes. For logical operations, we have `y ⩒= f(args...)` self reversible. For other arithmetic operations, we regard `y += f(args...)` and `y -= f(args...)` are reversible to each other. Here `f` can be `identity`, `*`, `/` and `^` et. al. Besides the above two types of operations, `SWAP` operation that exchanges the contents in two memory spaces

are also widely used in reversible computing systems. Here, it is worth noticing that += and -= are not exactly reversible to each other because floating point number operations have the rounding error. For applications sensitive to rounding errors, we should consider using other number systems, which will be discussed in Sec. V B

B. NiLang's Reversible IR

In the last subsection, we have reviewed basic building blocks of a typical reversible language. In order to insert the code of obtaining gradients into the reversed program, the reversible language design should have related abstraction power. It can be achieved by utilizing the multiple dispatch of Julia. We can wrap a number with a special type with gradient field called `GVar` and dispatch it to instructions that update gradient fields for it. Similar design is called `Dual` number in the tangent mode AD package ForwardDiff [40].

The main feature of NiLang is contained in a single macro `@i` that compiles a reversible function. The allowed statements in this eDSL are shown in Appendix A. The following is a minimal example of compiling a NiLang function to native julia function.

```
julia> using NiLangCore, MacroTools

julia> macroexpand(Main, :(@i function f(x, y)
           SWAP(x, y)
       end)) |> MacroTools.prettify
quote
    $(Expr(:meta, :doc))
    function $(Expr(:where, :(f(x, y))))
        gaur = SWAP(x, y)
        x = (NiLangCore.wrap_tuple(gaur))[1]
        y = (NiLangCore.wrap_tuple(gaur))[2]
        return (x, y)
    end
    if typeof(f) != typeof(~f)

        function $(Expr(:where, :((  #$ TODO: remove this comment
                    mongoose::typeof(~f))(x, y))))
            mandrill = (~SWAP)(x, y)
            x = (NiLangCore.wrap_tuple(mandrill))[1]
            y = (NiLangCore.wrap_tuple(mandrill))[2]
            return (x, y)
        end
    end
    if !(NiLangCore._hasmethod1(
                NiLangCore.isreversible, typeof(f)))
        NiLangCore.isreversible(::typeof(f)) = true
    end
end
```

Macro `@i` generates three functions `f`, `~f` and `NiLangCore.isreversible`. `f` and `~f` are a pair of functions that are reversbile to each other, where `~f` is an callable of type `Inv{typeof(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, in other words, it simulates a function that modifies inputs inplace. At the end this function, this macro attaches a return statement that returns all input variables. Finally, it defines `NiLangCore.isreversible` for `f` to mark it as reversible.

To understand the design of reversibility, we first introduce a reversible IR that plays a central role in NiLang. In this IR, a statement can be an instruction, a function call, a control flow, a memory allocation/deallocation, or the inverse statement "~". Any statement is this IR has a unique inverse as shown in Table I.

statement	**inverse**
<f>(<args>...)	(~<f>)(<args>...)
<y> += <f>(<args>...)	<y> -= <f>(<args>...)
<y> .+= <f>.(<args>...)	<y> .-= <f>.(<args>...)
<y> ⊻= <f>(<args>...)	<y> ⊻= <f>(<args>...)
<y> .⊻= <f>(<args>...)	<y> .⊻= <f>(<args>...)
<a> ← <expr>	<a> → <expr>
(<T1> => <T2>)(<x>)	(<T2> => <T1>)(<x>)
begin <stmts> end	begin ~(<stmts>) end

if (<pre>, <post>)
  <stmts1>
else
  <stmts2>
end
	if (<post>, <pre>)
	  ~(<stmts1>)
	else
	  ~(<stmts2>)
	end

while (<pre>, <post>)
  <stmts>
end
	while (<post>, <pre>)
	  ~(<stmts>)
	end

for <i>=<m>:<s>:<n> <stmts> end	for <i>=<m>:-<s>:<n> ~(<stmts>) end
@safe <expr>	@safe <expr>

Table I: A collection of reversible statements, where left side and right side are reversible to each other. "." is the symbol for broadcasting magic in Julia, "~" is the symbol for reversing a statement or a function. <pre> stands for precondition, <post> stands for postcondition, <args>... stands for the argument list of a function, <stmts> stands for statement, <exprs> stands for expression, <T1> and <T2> stand for types.

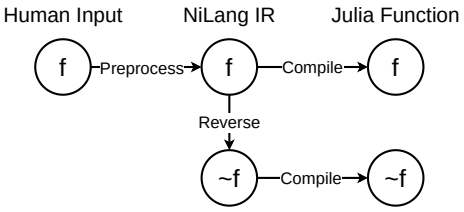
,

"←" and "→" are symbols for memory allocation and deallocation, one can input them by typing "\leftarrow" and "\rightarrow" respectively followed by a Tab key in a Julia editor or REPL. "`begin <stmts> end`" is the block statement in Julia, it represents a code block. It can be inverted by reversing the order as well as each element in it. A `if` or `while` statement is similar to its native Julia counterpart except the conditional expression in statements is a tuple of precondition and postcondition. Finally, the special macro `@safe` allows users to use external statements that do not

break reversibility. For example, one can use `@safe @show <var>` for debugging.

C. Compiling

The compilation of a reversible function contains three stages.



The first stage pre-processes human inputs to a reversible IR. The preprocessor expands the symbol "~" in the post-condition field of `if` statement by copying the precondition, adds missing ancilla "←" statements to ensure "←" and "→" appear in pairs inside a function, a `while` statement or a `for` statement, and expands the uncomputing macro `~@routine`. Since the compmute-copy-uncompute paradigm is extensively used in reversible programming for uncomputing ancillas. One can use `@routine <stmt>` statement to record a statement, and `~@routine` to insert `~<stmt>` for uncomputing. The following example pre-processes an `if` statement to the reversible IR.

```
julia> using NiLangCore, MacroTools

julia> NiLangCore.precom_ex(
       :(if (x > 3, ~)
            @routine z += x * y
            ~@routine
       end), NiLangCore.PreInfo()
       ) |> MacroTools.prettify
:(if (x > 3, x > 3)
      z += x * y
      z -= x * y
  else
  end)
```

In this example, since the precondition "`x > 3`" is not change after execution of the specific branch, we omit the postcondition by putting a "~" in this field. "`@routine`" records a statement, the statement can also be a "`begin <stmts> end`" block as a sequence of statements.

The second stage generates the reversed code according to table Table I. For example,

```
julia> NiLangCore.dual_ex(
       :(if (pre, post)
            z += x * y
       else
            z += x / y
       end)
       ) |> MacroTools.prettify
:(if (post, pre)
      z -= x * y
  else
      z -= x / y
  end)
```

The third stage is translating this IR and its inverse to native Julia code. It explains all functions as inplace and insert codes about reversibility check. At the end of a function definition, it attaches a return statement that returns all input arguments. After this, the function is ready to execute on the host language. The following example shows how an `if` statement is transformed in this stage.

```
julia> NiLangCore.compile_ex(
       :(if (pre, post)
            z += x * y
       else
            z += x / y
       end), NiLangCore.CompileInfo()
       ) |> MacroTools.prettify
quote
    bat = pre
    if bat
        @assignback (PlusEq(*))(z, x, y)
    else
        @assignback (PlusEq(/))(z, x, y)
    end
    @invcheck post bat
end
```

The compiler translates the instruction according to Table III and adds `@assignback` before each instruction and function call statement. The macro `@assignback` assigns the output of a function back to the arguments of that function. `@invcheck post bat` checks the consistency between preconditions and postconditions to ensure reversibility. This statement will throw an `InvertibilityError` error if target variables `bat` and `post` are not "equal" to each other up to a certian torlerance.

D. Types and Dataviews

So far, the language design is not too different from a traditional reversible language. To implement the adjoint mode AD, we introduce types and dataviews. The type that used in the reversible context is just a normal Julia type with an extra requirement of having reversible constructors.

The inverse of a constructor is called a "destructor", which unpacks data and deallocates derived fields. A reversbile constructor is implemented by reinterpreting the `new` function in Julia. Creating a new object is similar to creasting an ancilla. Let's consider the following statement.

```
x ← new{TX, TG}(x, g)
```

It works just like an ancilla, except it deallocates `g` directly at the same time. This makes sense because `new` is special that the its output keeps all information of its arguments, all input variables that does not appear in the output can be discard safely. Its inverse is

```
x → new{TX, TG}(x, g)
```

It unpacks structure `x` and assign fields to corresponding variables in argument list. The following example shows a non-complete definition the reversible type `GVar`.

```
julia> using NiLangCore

julia> @i struct GVar{T,GT} <: IWrapper{T}
           x::T
           g::GT
           function GVar{T,GT}(x::T, g::GT) where
                                       {T,GT}
               new{T,GT}(x, g)
           end
           function GVar(x::T, g::GT) where {T,GT}
               new{T,GT}(x, g)
           end
           @i function GVar(x::T) where T
               g ← zero(x)
               x ← new{T,T}(x, g)
           end
           @i function GVar(x::AbstractArray)
               GVar.(x)
           end
       end

julia> GVar(0.5)
GVar{Float64,Float64}(0.5, 0.0)

julia> (~GVar)(GVar(0.5))
0.5

julia> (~GVar)(GVar([0.5, 0.6]))
2-element Array{Float64,1}:
 0.5
 0.6
```

`GVar` has two fields, they correspond to the value and gradient of a variable. Here, we put `@i` macro before both `struct` and `function` statements. The ones before functions generates forward and backward functions, while the one before `struct` moves `~GVar` functions to the outside of the type definition, otherwise the inverse function will be ignored by Julia compiler.

Since an operation changes data inplace in NiLang, hence a field of a immutable instance should also be "modifiable". Let us first consider the following example.

```
julia> arr = [GVar(3.0), GVar(1.0)]
2-element Array{GVar{Float64,Float64},1}:
 GVar{Float64,Float64}(3.0, 0.0)
 GVar{Float64,Float64}(1.0, 0.0)

julia> x, y = 1.0, 2.0
(1.0, 2.0)

julia> @instr -arr[2].g += x * y
2.0

julia> arr
2-element Array{GVar{Float64,Float64},1}:
 GVar{Float64,Float64}(3.0, 0.0)
 GVar{Float64,Float64}(1.0, -2.0)
```

In Julia language, the assign statement above will throw a syntax error because the function call "-" can not be assigned, and `GVar` is an immutable type. In NiLang, we use the macro `@assignback` to modify an immutable data directly. It translates the above statement to

```
1 res = (PlusEq(*))(-arr[2].g, x, y)
2 arr[2] = chfield(arr[2], Val(:g),
3     chfield(arr[2].g, -, res[1]))
4 x = res[2]
5 y = res[3]
```

The first line `PlusEq(*)(-arr[3].g, x, y)` computes the output as a tuple of length 3. At lines 2-3, `chfield(x, Val{:g}, val)` modifies the `g` field of `x` and `chfield(x, -, res[1])` returns `-res[1]`. Here, modifying a field requires the default constructor of a type not overwritten. The assignments in lines 4 and 5 are straightforward. We call a bijection of a field of an object a "dataview" of this object, it is directly modifiable in NiLang. The definition of dataview can be found in Appendix A.

III. REVERSIBLE AUTOMATIC DIFFERENTIATION

Local Jacobians and Hessians for basic instructions used in this section could be found in Appendix B 1.

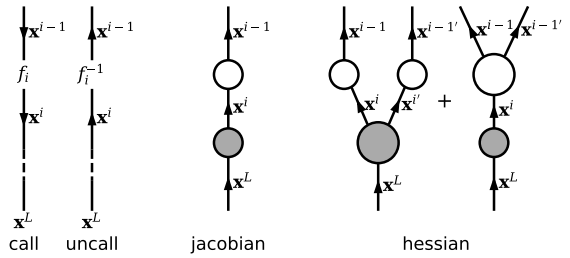


Figure 3: Computational processes in the tensor network diagram, a big circle with three legs represents a Hessian, a small circle with two legs represents a Jacobian. Dangling edges and connected edges stands for unpaired and paired labels respectively in the Einstein's notation. From left to right, the diagrams represent computing, uncomputing, back propagating Jacobians and back propagating Hessians.

A. First order gradient

Consider a computation $\mathbf{x}^{i-1} = f_i^{-1}(\mathbf{x}^i)$ in a reversed program, the Jacobians can be propagated in the reversed direction like

$$\begin{aligned} J^{\mathbf{x}^L}_{\mathbf{x}^{L'}} &= \delta_{\mathbf{x}^L,\mathbf{x}^{L'}}, \\ J^{\mathbf{x}^L}_{\mathbf{x}^{i-1}} &= J^{\mathbf{x}^L}_{\mathbf{x}^i} J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}}, \end{aligned} \tag{1}$$

where $\mathbf{x}^L$ represents the outputs of the program. In adjoint mode AD, it is a scalar. $J^{\mathbf{x}^L}_{\mathbf{x}^i} \equiv \frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^i}$ is the Jacobian to be propagated, and $J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}}$ is the local Jacobian matrix. The Einstein's notation [41] is used here so that the duplicated index $\mathbf{x}^i$ in the second line is summed over. Eq. (1) can be rewritten in the diagram of tensor networks [42] as shown in Fig. 3. The algorithm to compute the adjoint mode AD can be summarized as follows.

Algorithm 1: Reversible programming AD

Result: `grad`.(**x**)
let `iloss` be the index of the loss in **x**
x ← f(**x**)
for k = *1:length*(**x**) **do**
  $\mathbf{x}_k \leftarrow$ `GVar`($\mathbf{x}_k$, $\delta_{k,\text{iloss}}$)
x ← f^{-1}(**x**)

We first compute the results with the forward pass $f(x)$. Then we wrap each output with a gradient field. The gradient field is initialized to 1 if the variable is the loss else 0. Then we run the backward pass $f^{-1}(x)$ to update the gradient fields of variables. The gradients can be accessed using the `grad` dataview of output variables. The computation of gradients are implemented with multiple dispatch, that is, when an instruction meets a `GVar` instance, it calls a different routine. For example, the backward rules for instructions $\oplus(*)$ and $\ominus(*)$ can be defined by overloading **either** of them as following.

```
@i function ⊖(*)(out!::GVar, x::GVar, y::GVar)
    value(out!) -= value(x) * value(y)
    grad(x) += grad(out!) * value(y)
    grad(y) += value(x) * grad(out!)
end
```

The backward rule defined on $\ominus(*)$ will be used in the back-propagation of $\oplus(*)$. The compiler also binds the backward rules on $\oplus(*)$ automatically. This is doable because taking inverse and computing gradients commute to each other [43]. One can check the correctness of this definition easily as follows.

```
julia> using NiLang, NiLang.AD

julia> a, b, y = GVar(0.5), GVar(0.6), GVar(0.9)
(GVar(0.5, 0.0), GVar(0.6, 0.0), GVar(0.9, 0.0))

julia> @instr grad(y) += identity(1.0)

julia> @instr y += a * b
GVar(0.6, -0.5)

julia> a, b, y
(GVar(0.5, -0.6), GVar(0.6, -0.5), GVar(1.2, 1.0))

julia> @instr y -= a * b
GVar(0.6, 0.0)

julia> a, b, y
(GVar(0.5, 0.0), GVar(0.6, 0.0), GVar(0.899999, 1.0))
```

Here, since $J(\ominus(*)) = J(\oplus(*))^{-1}$, consecutively applying them will restore the gradient fields of all variables. The overhead of using `GVar` type is negligible thanks to Julia's multiple-dispatch and type inference.

[**JG: Grammarly here!**]

B. Second-order gradient

Second-order gradients can be obtained in three different approaches, forward differentiating the adjoint program, adjoint differentiating the adjoint program and Hessian propagation.

1. *Forward differentiating the adjoint program*

Combining the adjoint program in NiLang with dual numbers is a simple yet efficient way to obtain Hessians. By wrapping the elementary type with `Dual` that defined in package ForwardDiff [40] and throwing it into the gradient program defined in NiLang, one obtains one row/column of the Hessian matrix straightforward. We will show an example of using

forward differentiating in Newton trust region optimization in Sec. IV E.

2. *Adjoint differentiating the adjoint program*

Back propagating first-order gradients is also widely used to obtain the second-order gradients. Suppose the function space is closed under gradient operation, one can obtain higher-order gradients by recursively differentiating lower order gradient functions without defining new backward rules. An example is shown in Appendix G.

3. *Hessian propagation*

It is also possible to back-propagate Hessians directly [44] using the relation

$$\begin{aligned} H^{\mathbf{x}^L}_{\mathbf{x}^{L'},\mathbf{x}^{L''}} &= \mathbf{0}, \\ H^{\mathbf{x}^L}_{\mathbf{x}^{i-1},\mathbf{x}^{i-1'}} &= J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}} H^{\mathbf{x}^L}_{\mathbf{x}^i,\mathbf{x}^{i'}} J^{\mathbf{x}^{i'}}_{\mathbf{x}^{i-1'}} + J^{\mathbf{x}^L}_{\mathbf{x}^i} H^{\mathbf{x}^i}_{\mathbf{x}^{i-1},\mathbf{x}^{i-1'}}. \end{aligned} \tag{2}$$

Here, the Hessian is represented as a tensor $H^{\mathbf{x}^L}_{\mathbf{x}^{i-1},\mathbf{x}^{i-1'}}$ of rank three. In tensor network diagram, the above equation can be represented as the right panel of Fig. 3. The advantange of using Hessian propagation is it can be generalized to Taylor propagation in the future. With respect to the order of gradients, Taylor propagation is exponentially more efficient in obtaining higher-order gradients than differentiating lower order gradients recursively. Comparing with operator overloading, source to source automatic differentiation has the advantage of having very limited primitives, exhausted support to Hessian propagation is possible. An example to obtain Hessians is provided in Sec. IV A.

C. Gradient on ancilla problem

In this section, we introduced an easily overlooked problem in our reversible AD framework. An ancilla sometimes can carry a nonzero gradient when it is going to be deallocated. As a result, even if an ancilla can be uncomputed rigorously in the original program, its `GVar` wrapped version is not necessarily safely deallocated. In NiLang, we simply "drop" the gradient field of ancillas instead of raising an error. In the following, we justify our decision by proving the following theorem

***Theorem* 1.** *Deallocating an ancilla with constant value field and nonzero gradient field does not harm the reversibility of a function.*

Proof. Consider a reversible function $\mathbf{x}^i, b = f_i(\mathbf{x}^{i-1}, a)$, where a and b are the input and output values of an ancilla. Since both a, b are constants that are indepedent of input $\mathbf{x}^{i-1}$, we have

$$\frac{\partial b}{\partial \mathbf{x}^{i-1}} = \mathbf{0}. \tag{3}$$

Discarding gradients should not have any effect to the `value` fields of outputs. The key is to show `grad(b)` $\equiv \frac{\partial \mathbf{x}^L}{\partial b}$ does appear in the `grad` fields of the output. It can be seen from the back-propagation rule

$$\frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^{i-1}} = \frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^i}\frac{\partial \mathbf{x}^i}{\partial \mathbf{x}^{i-1}} + \frac{\partial \mathbf{x}^L}{\partial b}\frac{\partial b}{\partial \mathbf{x}^{i-1}}, \tag{4}$$

where the second term with $\frac{\partial \mathbf{x}^L}{\partial b}$ vanishes naturally. □

IV. EXAMPLES

In this section, we introduce several examples. We will discuss the first example, the first kind Bessel function, in most detail. We compare the difference between the irreversible and reversible implementations of this function, as well as the difference between normal computational graph and memory oriented computational graph. Then we show how to obtain first and second order gradients automatically in the reversible AD framework. We benchmark different source to source AD implementations of the Bessel function. Then we show how to differentiate sparse matrix operations, unitary matrix multiplication and QR decomposition. Finally, we show how to solve the graph embedding problem variationally with trust region Newton method, a second order optimizer that requires Hessians.

A. The first kind Bessel function

An Bessel function of the first kind of order ν can be computed using Taylor expansion

$$J_\nu(z) = \sum_{n=0}^{\infty} \frac{(z/2)^\nu}{\Gamma(k+1)\Gamma(k+\nu+1)}(-z^2/4)^n \tag{5}$$

where $\Gamma(n) = (n-1)!$ is the Gamma function. One can compute the accumulated item iteratively as $s_n = -\frac{z^2}{4}s_{n-1}$. The irreversible implementation is

```
function besselj(ν, z; atol=1e-8)
    k = 0
    s = (z/2)^ν / factorial(ν)
    out = s
    while abs(s) > atol
        k += 1
        s *= (-1) / k / (k+ν) * (z/2)^2
        out += s
    end
    out
end
```

This computational process could be diagrammatically represented as a DAG as show in Fig. 4(a). In this diagram, the data is represented as an edge. It connects at most two nodes, one generates this data and one consumes it. Computational

graph is more likely a mathematical expression, it can not describe inplace functions or control flows conveniently because it does not have the notation for memory and loops.

In the following, we introduce the reversible implementation and the memory oriented computational graph. The above bessel function contains a loop with irreversible "*=" operation inside. Intuitively, consequtive multiplication requires an increasing size of tape to cache the intermediate state s_n, since one can not release state s_{n-1} directly after computing s_n [23]. To reduce the memory allocation without increasing the time complexity of program, we introduce the following reversible approximate multiplier

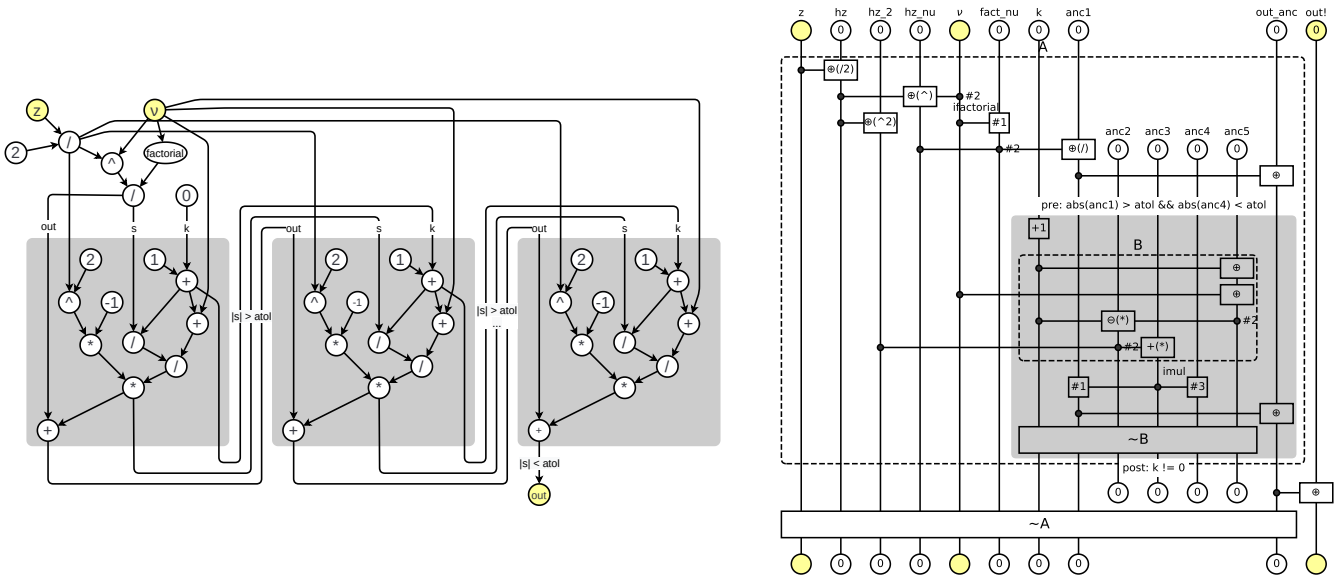
```
1 @i @inline function imul(out!, x, anc!)
2     anc! += out! * x
3     out! -= anc! / x
4     SWAP(out!, anc!)
5 end
```

Here, instruction `SWAP` exchanges values of two variables, `anc!` ≈ 0 is a *dirty ancilla*. Line 2 computes the result and accumulates it to the dirty ancilla, we get an approximately correct output in `anc!`. Line 3 uncompute `out!` approximately by using the information stored in `anc!`, leaving a dirty zero state in register `out!`. Line 4 swaps the contents in `out!` and `anc!`. Finally, we have an approximately correct output and a dirtier ancilla. This "approximate uncomputing" trick can be extensively used in practise. It mitigated the artificial irreversibility brought by the number system that we have adopted at the cost of output precision. The reason why this trick works here lies in the fact that from the mathematic perspective the state in nth step $\{s_n, z\}$ contains the same amount of information as its previous state $\{s_{n-1}, z\}$ except some special points, it is highly possible to find an equation to uncompute the previous state from the current state. With this approximate multiplier, we implement J_ν as follows.

```
using NiLang, NiLang.AD

@i function ibesselj(out!, ν, z; atol=1e-8)
    k ← 0
    fact_nu ← zero(ν)
    halfz ← zero(z)
    halfz_power_nu ← zero(z)
    halfz_power_2 ← zero(z)
    out_anc ← zero(z)
    anc1 ← zero(z)
    anc2 ← zero(z)
    anc3 ← zero(z)
    anc4 ← zero(z)
    anc5 ← zero(z)

    @routine begin
        halfz += z / 2
        halfz_power_nu += halfz ^ ν
        halfz_power_2 += halfz ^ 2
        ifactorial(fact_nu, ν)

        anc1 += halfz_power_nu/fact_nu
        out_anc += identity(anc1)
        while (abs(unwrap(anc1)) > atol &&
                abs(unwrap(anc4)) < atol, k!=0)
            k += identity(1)
            @routine begin
                anc5 += identity(k)
                anc5 += identity(ν)
                anc2 -= k * anc5
                anc3 += halfz_power_2 / anc2
            end
            imul(anc1, anc3, anc4)
            out_anc += identity(anc1)
            ~@routine
        end
    end
    out! += identity(out_anc)
    ~@routine
end

@i function ifactorial(out!, n)
    out! += identity(1)
    for i=1:n
        mulint(out!, i)
    end
end
```

Here, only a constant number of ancillas are used, while the time complexity does not increase comparing to its irreversible counterpart. Ancilla `anc4` plays the role of *dirty ancilla* in multiplication, it is uncomputed rigorously in the uncomputing stage marked by `~@routine`.

This reversible program can be diagramatically represented as a memory oriented computational graph as show in Fig. 4 (b). In this graph, a variable is a vertical line, a function is a parallel line. The key difference to traditional computational graph is it adopts a variable centric view. A variable can be used by multiple functions, i.e. it is a hypergraph rather than a simple graph. If a function uses a variable but does not change the contents in it, we call this variable a control parameter of this function and put a dot at the cross. Otherwise if the



(a) Where a vertex (circle) is an operation and a directed edge is a variable. The gray region is the body of while loop. The while loop is unrolled.

(b) Vertical lines are variables, boxes and dots connected by parallel lines are operations. A dot represents using a variable as control while a box represents mutating a variable. The gray box represents the body of a while loop, where a precondition and postcondition are positioned on top and bottom of this box.

Figure 4: The comparison between (a) tranditional and (b) memory oriented computational graph of the first kind Bessel function.

content is changed, we put a square. This diagram can be used to analyse uncomputable variables. In this example routine "B" uses `hz_2`, ν and `k` as control parameters, and changes the contents in `anc2`, `anc3` and `anc5`. while the following operation `imul` does not change these variables. Hence we can apply the inverse routine `~B` to safely restore contents in `anc2`, `anc3` and `anc5`, this is what people called compute-copy-uncompute paradigm.

One can obtain gradient of this function by calling `ibesselj'`

```
julia> out!, x = 0.0, 1.0
(0.0, 3.0)

julia> ibesselj'(Val(1), out!, 2, x)
(GVar(0.0, 1.0), 2, GVar(3.0, 0.0149981304))
```

The Hessian can be obtained by feeding dual numbers into this gradient function.

```
julia> using ForwardDiff: Dual
julia> _, hxout!, _, hxx = ibesselj'(Val(1),
        Dual(out!, zero(out!)), 2, Dual(x, one(x)))
julia> grad(hxx).partials[1]
```

Here, the gradient field of `hxx` is defined as $\frac{\partial \text{out!}}{\partial x}$, which is a Dual number. It has a field partials that stores the derivative with respect to `x`. It corresponds to the Hessian $\frac{\partial \text{out!}^2}{\partial x^2}$ that we need. See Appendix G for alternative approaches to obtain its Hessian.

1. *Benchmark*

We have different source to source automatic differentiation implementations of the first type Bessel function and show the benchmarks below.

In Table II, Julia is the CPU time used for running the irreversible forward program, it is the baseline for benchmarking. NiLang (call/uncall) is the time of reversible call or uncall, both of them are 2.8 times slower than its irreversible counterpart. Here, we have remove the reversibility check, otherwise it will take more time than this. ForwardDiff gives the best performance, it is even faster than manually derived

	Tangent/Adjoint	$T_{\min}$**/ns**	Space/KB
Julia	-	22	0
NiLang (call/uncall)	-	61	0
ForwardDiff	Tangent	35	0
Manual	Adjoint	83	0
NiLang.AD	Adjoint	271	0
NiLang.AD (GPU)	Adjoint	1.4	0
Zygote	Adjoint	31201	13.47
Tapenade	Adjoint	?	?

Table II: Time and space used in computing the gradients of Bessel functions $J_2(1.0)$. The CPU device is Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz and the GPU device is Nvidia Titan V. The GPU time is estimated by broadcasting the gradient function on CUDA array of size 2^{17} and take average.

gradients

$$J'_v(z) = \frac{J_{v-1} - J_{v+1}}{2}. \tag{6}$$

Except an effient implementation, its high performance also lies in the fact that this functions is suited for tangent mode AD which has single input. NiLang.AD represents the reversible AD in NiLang, it takes 12.3 times the original program. Still, it is much faster than naive checkpointing. In Zygote, the memory allocation are not all from checkpointing. Some of them are related to type instability. It uses closure to handle checkpointing, where external type instability will cause type instability of interia. Tapenade [**JG: needs your effort, @MLS**]

B. Sparse Matrices

Source to source automatic differentiation is very useful in differentiating sparse matrices. It is a well known problem that sparse matrix operations can not benefit directly from generic backward rules that defined for dense matrix, because generic rules does not keep the sparse structure.

In the following, we will show reversible AD can differentiate the Frobenius dot product between two sparse matrices with state of the art performance. Here, the Frobenius dot product is defined as `trace(A'B)`. This code is adapted from the irreversible implementation in Julia package `SparseArrays`.

```
@i function dot(r::T, A::SparseMatrixCSC{T},
        B::SparseMatrixCSC{T}) where {T}
    m ← size(A, 1)
    n ← size(A, 2)
    @invcheckoff branch_keeper ← zeros(Bool, 2*m)
    @safe size(B) == (m,n) ||
        throw(DimensionMismatch(
        "matrices must have the same dimensions"))
    @invcheckoff @inbounds for j = 1:n
        ia1 ← A.colptr[j]
        ib1 ← B.colptr[j]
        ia2 ← A.colptr[j+1]
        ib2 ← B.colptr[j+1]
        ia ← ia1
        ib ← ib1
        @inbounds for i=1:ia2-ia1+ib2-ib1-1
            ra ← A.rowval[ia]
            rb ← B.rowval[ib]
            if (ra == rb, ~)
                r += A.nzval[ia]' * B.nzval[ib]
            end
            # b move -> true, a move -> false
            branch_keeper[i] ⊻= ia == ia2-1 ||
                ra > rb
            ra → A.rowval[ia]
            rb → B.rowval[ib]
            if (branch_keeper[i], ~)
                ib += identity(1)
            else
                ia += identity(1)
            end
        end

        ~@inbounds for i=1:ia2-ia1+ib2-ib1-1
            # b move -> true, a move -> false
            branch_keeper[i] ⊻= ia == ia2-1 ||
                    A.rowval[ia] > B.rowval[ib]
            if (branch_keeper[i], ~)
                ib += identity(1)
            else
                ia += identity(1)
            end
        end
    end
    @invcheckoff branch_keeper → zeros(Bool, 2*m)
end
```

With simple adaptation, the code becomes reversible. Here, the key point is using a `branch_keeper` vector to cache branch decisions. From the benchmark, we show time costs is approximately 1.6 times the Julia native forward implementation.

```
julia> @benchmark SparseArrays.dot($a, $b)
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     94.537 μs (0.00% GC)
  median time:      96.959 μs (0.00% GC)
  mean time:        98.188 μs (0.00% GC)
  maximum time:     189.291 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1

julia> out! = SparseArrays.dot(a, b)
25.19659286755114

julia> @benchmark (~dot)($(GVar(out!, 1.0)),
       $(GVar.(a)), $(GVar.(b)))
BenchmarkTools.Trial:   #$ TODO: remove this comment
  memory estimate:  2.17 KiB
  allocs estimate:  2
  --------------
  minimum time:     151.392 μs (0.00% GC)
  median time:      153.153 μs (0.00% GC)
  mean time:        155.884 μs (0.00% GC)
  maximum time:     270.534 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

Here, we turn off the irreversibility check off to achieve best performance. One can always turn off this check after the debugging. It is also possible to implement sparse matrix multiplication and other operations reversibly.

C. Unitary Matrices

A unitary matrices features uniform eigenvalues and reversibility. It is widely used as an approach to ease the gradient exploding and vanishing problem [45–47] and the memory wall problem [8]. One of the simplest way to parametrize a unitary matrix is representing a unitary matrix as a product of two-level unitary operations [47]. A real unitary matrix of size N can be parametrized compactly by $N(N-1)/2$ rotation operations [48]

$$\mathrm{ROT}(a!, b!, \theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} a! \\ b! \end{pmatrix}, \tag{7}$$

where θ is the rotation angle, `a!` and `b!` are target registers.

```
using NiLang, NiLang.AD

@i function umm!(x!, θ)
    @safe @assert length(θ) ==
            length(x!)*(length(x!)-1)/2
    k ← 0
    for j=1:length(x!)
        for i=length(x!)-1:-1:j
            k += identity(1)
            ROT(x![i], x![i+1], θ[k])
        end
    end

    k → length(θ)
end
```

Here, the ancilla `k` is deallocated manually by specifying its value, because we know the loop size is $N(N-1)/2$. We define the test functions in order to check gradients.

```
julia> @i function isum(out!, x::AbstractArray)
           for i=1:length(x)
               out! += identity(x[i])
           end
       end

julia> @i function test!(out!, x!::Vector, θ::Vector)
           umm!(x!, θ)
           isum(out!, x!)
       end

julia> out, x, θ = Loss(0.0), randn(4), randn(6);

julia> @instr test!'(out, x, θ)

julia> x
4-element Array{GVar{Float64,Float64},1}:
 GVar(1.220182125326287, 0.14540743042341095)
 GVar(2.1288634811475937, -1.3749962375499805)
 GVar(1.2696579252569677, 1.42868739498625)
 GVar(0.1083891125379283, 0.2170123344615735)

julia> @instr (~test!')(out, x, θ)

julia> x
4-element Array{Float64,1}:
 1.220182125326287
 2.1288634811475933
 1.2696579252569677
 0.10838911253792821
```

In the above testing code, `test'` attaches a gradient field to each element of `x`. `~test'` is the inverse program that erase the gradient fields. Notably, this reversible implementation costs zero memory allocation although it changes the target variables inplace.

D. QR decomposition

Let's consider a naive implementation of QR decomposition from scratch. We admit this implementation is just a proof of principle which does not even consider reorthogonalization.

```
using NiLang, NiLang.AD

@i function qr(Q, R, A::Matrix{T}) where T
    anc_norm ← zero(T)
    anc_dot ← zeros(T, size(A,2))
    ri ← zeros(T, size(A,1))
    for col = 1:size(A, 1)
        ri .+= identity.(A[:,col])
        for precol = 1:col-1
            dot(anc_dot[precol], Q[:,precol], ri)
            R[precol,col] +=
                identity(anc_dot[precol])
            for row = 1:size(Q,1)
                ri[row] -=
                    anc_dot[precol] * Q[row, precol]
            end
        end
        norm2(anc_norm, ri)

        R[col, col] += anc_norm^0.5
        for row = 1:size(Q,1)
            Q[row,col] += ri[row] / R[col, col]
        end

        ~begin
            ri .+= identity.(A[:,col])
            for precol = 1:col-1
                dot(anc_dot[precol], Q[:,precol], ri)
                for row = 1:size(Q,1)
                    ri[row] -= anc_dot[precol] *
                        Q[row, precol]
                end
            end
            norm2(anc_norm, ri)
        end
    end
end
```

Here, in order to avoid frequent uncomputing, we allocate ancillas `ri` and `anc_dot` as vectors. The expression in `~` is used to uncompute `ri`, `anc_dot` and `anc_norm`. `dot` and `norm2` are reversible functions to compute dot product and vector norm. Their implementations could be found in Appendix D. One can easily check the correctness of the gradient function

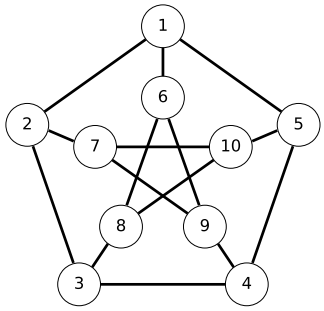


Figure 5: The Petersen graph with 10 vertices and 15 edges.

```
julia> A, q, r = randn(4,4), zero(A), zero(A);

julia> @i function test1(out, q, r, A)
           qr(q, r, A)
           isum(out, q)
       end

julia> check_grad(test1, (Loss(0.0), q, r, A))
true
```

Here, the loss function `test1` is defined as the sum of the output unitary matrix `q`. The `check_grad` function is a gradient checker function defined in module NiLang.AD.

E. Solving a graph embedding problem

Graph embedding has an interesting application of finding a representation for an order parameter [49] in condensed matter physics. Ref. 49 considers a problem of finding the minimum Euclidean space dimension k that a Petersen graph can fit into, with extra requirements that the distance between a pair of connected vertices has the same value l_1, the distance between a pair of disconnected vertices has the same value l_2 and $l_2 > l_1$. The Petersen graph is 10 vertices graph as shown in Fig. 5. Let's denote the set of connected and disconnected vertex pairs as L_1 and L_2 respectively. This problem can be variationally solved by differential programming by designing the following loss.

$$\begin{aligned}\mathcal{L} = &\text{Var}(\text{dist}(L_1)) + \text{Var}(\text{dist}(L_2)) \\ &+ \exp(\text{relu}(\text{E}(\text{dist}(L_1)) - \text{E}(\text{dist}(L_2)) + 0.1))) - 1\end{aligned} \tag{8}$$

The first line is a summation of distance variances in two sets of vertex pairs. The second line is used to guarantee $l_2 > l_1$.

```
using NiLang, NiLang.AD

@i function sqdistance(dist!, x1::AbstractVector{T},
        x2::AbstractVector) where T
    @inbounds for i=1:length(x1)
        x1[i] -= identity(x2[i])
        dist! += x1[i] ^ 2
        x1[i] += identity(x2[i])
    end
end

# bonds of a petersen graph
const L1 = [(1, 6), (2, 7), (3, 8), (4, 9), (5, 10),
    (1, 2), (2, 3), (3, 4), (4, 5), (1, 5), (6, 8),
    (8, 10), (7, 10), (7, 9), (6, 9)]

# disconnected bonds of a petersen graph
const L2 = [(1, 3), (1, 4), (1, 7), (1, 8), (1, 9),
    (1, 10), (2, 4), (2, 5), (2, 6), (2, 8), (2, 9),
    (2, 10), (3, 5), (3, 6), (3, 7), (3, 9), (3, 10),
    (4, 6), (4, 7), (4, 8), (4, 10), (5, 6), (5, 7),
    (5, 8), (5, 9), (6, 7), (6, 10), (7, 8), (8, 9),
    (9, 10)]

@i function embedding_loss(out!::T, x) where T
    v1 ← zero(T)
    m1 ← zero(T)
    v2 ← zero(T)
    m2 ← zero(T)
    diff ← zero(T)
    d1 ← zeros(T, length(L1))
    d2 ← zeros(T, length(L2))
    @routine begin
        for i=1:length(L1)
            sqdistance(d1[i],
                x[:,L1[i][1]],x[:,L1[i][2]])
        end
        for i=1:length(L2)
            sqdistance(d2[i],
                x[:,L2[i][1]],x[:,L2[i][2]])
        end
        var_and_mean_sq(v1, m1, d1)
        var_and_mean_sq(v2, m2, d2)
        m1 -= identity(m2)
        m1 += identity(0.1)
    end
    out! += identity(v1)
    out! += identity(v2)
    if (m1 > 0, ~)
        # to ensure mean(v2) > mean(v1)
        # if mean(v1)+0.1 - mean(v2) > 0, punish it.
        out! += exp(m1)
        out! -= identity(1)
    end
    ~@routine
end
```

One can access the gradient by typing

```
iloss'(Loss(0.0), randn(5, 10))
```

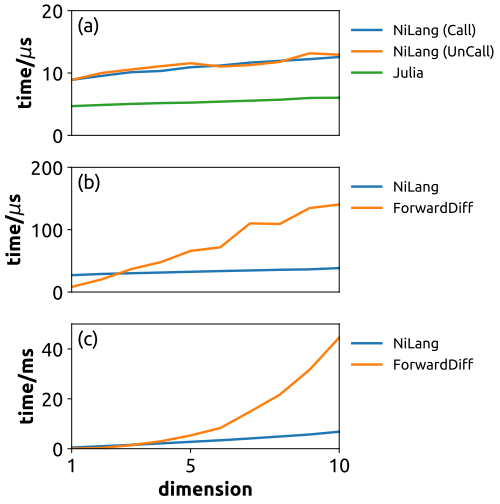


Figure 6: The time used to (a) call/uncall, (b) compute gradients and (c) compute Hessians.

We repeat the training for each dimension k from 1 to 9 and search for possible solutions by variationally optimizing the positions of vertices. In each training, we fixed two of the vertices and train the rest, otherwise the program will find the trivial solution with overlapped vertices. The training employs the Adam optimizer with a learning rate 0.01 [50], a 10000 steps training costs less than 1 second on a laptop. For $k = 5$, we can get a loss close to machine precision with high probability, while for $k < 5$, the loss is always much higher than 0. From the solution, it is easy to verify $l_2/l_1 = \sqrt{2}$.

1. *Benchmark*

Since the ForwardDiff itself provides the Hessian for users. It is interesting to benchmark how much performance can we get by forward differentiating an adjoint program than adjoint differentiating an adjoint program.

The benchmark uses the same setup as above. In this application, the number of input parameters scales as $10 \times k$, where k is the mebeding dimension of the graph. First, we see the reversible program is slower than native Julia program by a factor of ~ 2. However, the reversible program shows advantage in obtaining gradients when the dimension $k > 2$. The larger the number of input, the more advantage of adjoint mode AD due to the linear overhead in tangent mode AD. The same reason applies for computing Hessians, the mixed mode AD gives better performance when $k > 2$ comparing with pure tangent mode AD.

V. DISCUSSION AND OUTLOOK

In this paper, we show a program on an reversible Turing machine can be differentiated to any order reliably and efficiently without sophisticated designs to memorize com-

putational graph and intermediate states. We introduce a reversible Julia eDSL NiLang that implements a reversible AD. In a reversible programming language, we proposed to use "approximate uncomputing" trick to avoid the overhead of a reversible program in many practical cases. NiLang uses uncomputing to manage memory rather than closure, which does not frequently involve external variables in the function. Hence it is more friendly to type inference in Julia compiler. This explains why it performs so good in many cases.

In the following, we discussed some practical issues about reversible programming, and several future directions to go.

A. Time Space Tradeoff

In history, there has been many other interesting designs of reversible languages. However, current popular programming languages are all irreversible. In the simplest g-segment trade off scheme [51, 52], a RTM model has either a space overhead that proportional to computing time T or a computational overhead that sometimes can be exponential comparing with a irreversible counter part. The tradeoff between space and time is a crucial issue in the theory of RTM. In the following, we try to convince the readers that the overhead of reversible computing is not as terrible as people thought.

The overhead of reversing a program is bounded the checkpointing [53] strategy used in a traditional machine learning package that memorizes every inputs of primitives because similar strategy can also be used in reversible programming. [23] Reversible programming simply provides more alternatives to reduce the overhead. For example, the overhead in many iterative algorithms can often be removed with "arithmetic uncomputing" trick without sacrificing reversibility as shown in the `ibesselj` example in Sec. IV A.

Clever compiling can also be used to remove most overheads. Often, when we define a new reversible function, we allocate some ancillas at the beginning of the function and deallocate them through uncomputing at the end. The overhead comes from the uncomputing, in the worst case, the time used for uncomputing can be the same as the forward pass. In a hierarchical design, uncomputing can appear in every layer of abstraction. To quantify the overhead of uncomputing, we introducing the concept

***Definition* 1** (program granularity)**.** The log ratio between the execution time of a reversible program and its irreversible counter part

The computing time increases exponentially as the granularity increases. A cleverer compilation of a program can reduce the granularity by merging the uncomputing statements to avoid repeated efforts.

At last, making reversible programming an eDSL rather than an independent language allows flexible choices between reversibility and computational overhead. For example, in order to deallocate the gradient memory in a reversible language one has to uncompute the whole process of obtaining this gradient. In our eDSL, we can just deallocate the memory irreversibly, i.e. trade energy with time.

B. Instructions and Hardwares

So far, our eDSL is compiled to Julia. In the future, it can be compiled to reversible instructions [54] and executed on a reversible device. However, arithmetic instructions should be redesigned to support better reversible programs. The major obstacle to exact reversibility programming is current floating point adders and multipliers used in our computing devices are not exactly reversible. There are proposals of reversible floating point adders and multipliers [55–58], however these designs with allocate garbage bits in each operation. Alternatives include fixed point numbers [59] and logarithmic numbers [60, 61], where logarithmic number system is reversible under `*=` and `/=`. With these infrastructures, a reversible program can be executed without suffering from the rounding error.

Reversible instructions can be executed on an energy efficient reversible hardware. In the introduction, we mentioned several reversible hardwares. A reversible hardware can be those supporting reversible gates such as the Toffoli gate and the Fredkin gate, or like an adiabatic CMOS with the ability to recover signal energy. The latter is known as the generalized reversible computing. [62, 63] In the near future, there might be energy efficient artificial intelligence (AI) chips as coprocessors that our eDSL can compile to. Since reversible computing is mainly driven by quantum computing in recent years. In the following, we comment briefly on quantum devices.

1. *Quantum Computers*

Building a universal quantum computer [64] is difficult. The difficulty lies in the fact that it is extremely hard to protect a quantum state. Unlike a classical state, an quantum state is can not be cloned, meanwhile, it losses information by interacting with the environment, or decoherence. Classical reversible computing does not enjoy the quantum advantage, nor the quantum disadvantages of non-cloning and decoherence. It is technically more smooth to have a reversible computing device to bridge the gap between classical devices and universal quantum computing devices. By introducing entanglement little by little, we can accelerate some basic components in reversible computing. For example, quantum Fourier transformation provides an interesting alternative to the reversible adders and multipliers by introducing the CPHASE quantum gate. [65] The development of reversible compiling theory can be benefit quantum compiling directly.

C. Outlook

First of all, it can be used to generate AD rules for existing machine learning packages like Zygote.

Secondly, we can use it to overcome the memory wall problem in some applications. NiLang provides a systematic time-space trade off scheme through uncomputing. A successful related example is the memory efficient domain-specific AD

engine in quantum simulator Yao [8]. This domain-specific AD engine is written in a reversible style and solved the memory bottleneck in variational quantum simulations. It also gives hitherto the best performance in differentiating quantum circuit parameters. Similarity, we can write memory efficient normalizing flow [66] with NiLang. Normalizing flow is a successful class of generative model in both computer vision [67] and quantum physics [68, 69], where its building block bijector is reversible. We can use similar idea to differentiate reversible integrators [70, 71]. With reversible integrators, it should be possible to rewrite the control system in robotics [72] in a reversible style, where scalars are first class citizen rather than tensors. Writing a control program reversibly should boost the training performance a lot.

Thirdly, reversibility is a resource for training. For those who are interested in non-gradient based training. In Appendix C, we provide a self-consistency training strategy for reversible programs.

Lastly, the reversible IR is a good starting point to study quantum compiling. Most quantum programming language preassumes a classical coprocessor and use classical control flows [73] in universal quantum computing. However, we believe reversible control flows are also very important to a universal quantum computer.

To solve the above problems better, NiLang can be improved from multiple perspectives. Like we need a more efficient fixed point or log number system to avoid rounding errors. Currently we simulate reversible arithmetics with the julia fixed point number package. [59] Then we should optimize the compiling to decreases granularity of a program and reduces uncomputing overheads. There are also some known issues to be solved like the type inference problem, we have listed some of them on GitHub. These improvements needs participation of people from multiple fields.

VI. ACKNOWLEDGMENTS

Jin-Guo Liu thank Lei Wang for motivating the project with possible applications reversible integrator, normalizing flow and neural ODE. Xiu-Zhe Luo for discussion on the implementation details of source to source automatic differentiation, Shuo-Hui Li for helpful discussion on differential geometry. Damian Steiger for telling me the `comefrom` joke. Tong Liu and An-Qi Chen for helpful discussion on quantum adders and multipliers. Yin-Bo Ma for correcting typos by submmiting pull requests. Chris Rackauckas for helpful discussion on reversible integrator. Mike Innes for reviewing the comments about Zygote. Jun Takahashi for discussion about the graph embedding problem. Simon Byrne and Chen Zhao for helpful discussion on floating point and logarithmic numbers. The authors are supported by the National Natural Science Foundation of China under the Grant No. 11774398, the Strategic Priority Research Program of Chinese Academy of Sciences Grant No. XDB28000000.

[1] L. Hascoet and V. Pascual, ACM Transactions on Mathematical Software (TOMS) **39**, 20 (2013).
[2] 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).
[3] 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].
[4] 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.
[5] J.-G. L. Hao Xie and L. Wang, arXiv:2001.04121.
[6] G. H. Golub and C. F. Van Loan, *Matrix computations*, Vol. 3 (JHU press, 2012).
[7] H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, Physical Review X **9** (2019), 10.1103/physrevx.9.031041.
[8] 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].
[9] M. Seeger, A. Hetzel, Z. Dai, E. Meissner, and N. D. Lawrence, "Auto-differentiating linear algebra," (2017), arXiv:1710.08717 [cs.MS].
[10] Z.-Q. Wan and S.-X. Zhang, "Automatic differentiation for complex valued svd," (2019), arXiv:1909.02659 [math.NA].
[11] C. Hubig, "Use and implementation of autodifferentiation in tensor network methods with complex scalars," (2019), arXiv:1907.13422 [cond-mat.str-el].
[12] M. Innes, "Don't unroll adjoint: Differentiating ssa-form programs," (2018), arXiv:1810.07951 [cs.PL].
[13] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, CoRR **abs/1907.07587** (2019), arXiv:1907.07587.
[14] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," (2015), arXiv:1506.00019 [cs.LG].
[15] K. He, X. Zhang, S. Ren, and J. Sun, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016), 10.1109/cvpr.2016.90.
[16] M. MacKay, P. Vicol, J. Ba, and R. B. Grosse, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 9029–9040.
[17] D. Maclaurin, D. Duvenaud, and R. Adams, in *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 37, edited by F. Bach and D. Blei (PMLR, Lille, France, 2015) pp. 2113–2122.
[18] J. Behrmann, D. Duvenaud, and J. Jacobsen, CoRR **abs/1811.00995** (2018), arXiv:1811.00995.
[19] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, in *Advances in Neural Information Processing Systems 30*, edited

by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Curran Associates, Inc., 2017) pp. 2214–2224.
[20] J.-H. Jacobsen, A. W. Smeulders, and E. Oyallon, in *International Conference on Learning Representations* (2018).
[21] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, arXiv preprint arXiv:1209.5145 (2012).
[22] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM Review **59**, 65–98 (2017).
[23] K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).
[24] M. P. Frank, IEEE Spectrum **54**, 32–37 (2017).
[25] "MLStyle.jl," `https://github.com/thautwarm/MLStyle.jl`.
[26] C. Lutz, "Janus: a time-reversible language," (1986), *Letter to R. Landauer*.
[27] M. P. Frank, *The R programming language and compiler*, Tech. Rep. (MIT Reversible Computing Project Memo, 1997).
[28] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018).
[29] T. Haulund, "Design and implementation of a reversible object-oriented programming language," (2017), arXiv:1707.07845 [cs.PL].
[30] 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).
[31] J. G. Koller and W. C. Athas, in *Workshop on Physics and Computation* (1992) pp. 267–270.
[32] 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.
[33] K. Likharev, IEEE Transactions on Magnetics **13**, 242 (1977).
[34] V. K. Semenov, G. V. Danilov, and D. V. Averin, IEEE Transactions on Applied Superconductivity **13**, 938 (2003).
[35] R. Landauer, IBM journal of research and development **5**, 183 (1961).
[36] E. P. DeBenedictis, J. K. Mee, and M. P. Frank, Computer **50**, 76 (2017).
[37] D. R. Jefferson, ACM Transactions on Programming Languages and Systems (TOPLAS) **7**, 404 (1985).
[38] B. Boothe, ACM SIGPLAN Notices **35**, 299 (2000).
[39] C. H. Bennett (1973).
[40] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," (2016), arXiv:1607.07892 [cs.MS].
[41] Wikipedia contributors, "Einstein notation — Wikipedia, the free encyclopedia," (2020), [Online; accessed 20-February-2020].
[42] R. Orús, Annals of Physics **349**, 117–158 (2014).
[43] A. McInerney, *First steps in differential geometry* (Springer, 2015).
[44] J. Martens, I. Sutskever, and K. Swersky, "Estimating the hessian by back-propagating curvature," (2012), arXiv:1206.6464 [cs.LG].
[45] M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464.
[46] S. Wisdom, T. Powers, J. R. Hershey, J. L. Roux, and L. Atlas, "Full-capacity unitary recurrent neural networks," (2016), arXiv:1611.00035 [stat.ML].
[47] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231.
[48] C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).
[49] J. Takahashi and A. W. Sandvik, "Valence-bond solids, vestigial order, and emergent so(5) symmetry in a two-dimensional quantum magnet," (2020), arXiv:2001.10045 [cond-mat.str-el].
[50] D. P. Kingma and J. Ba, arXiv:1412.6980.
[51] C. H. Bennett, SIAM Journal on Computing **18**, 766 (1989).
[52] R. Y. Levine and A. T. Sherman, SIAM Journal on Computing **19**, 673 (1990).
[53] T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174.
[54] C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.
[55] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on Nanotechnology* (2010) pp. 233–237.
[56] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on Nanotechnology* (2011) pp. 451–456.
[57] T. D. Nguyen and R. V. Meter, "A space-efficient design for reversible floating point adder in quantum computing," (2013), arXiv:1306.3760 [quant-ph].
[58] T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, "Quantum circuits for floating-point arithmetic," (2018), arXiv:1807.02023 [quant-ph].
[59] "FixedPointNumbers.jl," `https://github.com/JuliaMath/FixedPointNumbers.jl`.
[60] F. J. Taylor, R. Gill, J. Joseph, and J. Radke, IEEE Transactions on Computers **37**, 190 (1988).
[61] "LogarithmicNumbers.jl," `https://github.com/cjdoris/LogarithmicNumbers.jl`.
[62] M. P. Frank, in *35th International Symposium on Multiple-Valued Logic (ISMVL'05)* (2005) pp. 168–185.
[63] M. P. Frank, in *Reversible Computation*, edited by I. Phillips and H. Rahaman (Springer International Publishing, Cham, 2017) pp. 19–34.
[64] M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," (2002).
[65] L. Ruiz-Perez and J. C. Garcia-Escartin, Quantum Information Processing **16** (2017), 10.1007/s11128-017-1603-1.
[66] I. Kobyzev, S. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," (2019), arXiv:1908.09257 [stat.ML].
[67] D. P. Kingma and P. Dhariwal, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 10215–10224.
[68] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real nvp," (2016), arXiv:1605.08803 [cs.LG].
[69] S.-H. Li and L. Wang, Physical Review Letters **121** (2018), 10.1103/physrevlett.121.260601.
[70] P. Hut, J. Makino, and S. McMillan, The Astrophysical Journal **443**, L93 (1995).
[71] D. N. Laikov, Theoretical Chemistry Accounts **137** (2018), 10.1007/s00214-018-2344-7.
[72] M. Giftthaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, Advanced Robotics **31**, 1225–1237 (2017).
[73] K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, and A. Paz, Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018 (2018), 10.1145/3183895.3183901.
[74] M. Bender, P.-H. Heenen, and P.-G. Reinhard, Rev. Mod. Phys. **75**, 121 (2003).

[75] T. Besard, C. Foket, and B. D. Sutter, CoRR **abs/1712.03112** (2017), arXiv:1712.03112.
[76] When people ask for the location in Beijing, they will start by asking which ring it is? We use the similar approach to locate the elements of Hessian matrix.

Appendix A: NiLang Grammar

To define a reversible function one can use "@i" plus a normal function definition like bellow

```
"""
docstring...
"""
@i function f(args..., kwargs...) where {...}
    <stmts>
end
```

where the definition of "<stmts>" are shown in the grammar on the next column. The following is a list of terminologies used in the definition of grammar

- *ident*, symbols
- *num*, numbers
- ϵ, empty statement
- *JuliaExpr*, native Julia expression
- [ ], zero or one repetitions.

Here, all *JuliaExpr* should be pure, otherwise the reversibility is not guaranteed. Dataview is a view of a data, it can be a bijective mapping of an object, an item of an array or a field of an object.

⟨Stmts⟩ ::= ϵ
| ⟨Stmt⟩
| ⟨Stmts⟩ ⟨Stmt⟩
⟨Stmt⟩ ::= ⟨BlockStmt⟩
| ⟨IfStmt⟩
| ⟨WhileStmt⟩
| ⟨ForStmt⟩
| ⟨InstrStmt⟩
| ⟨RevStmt⟩
| ⟨AncillaStmt⟩
| ⟨TypecastStmt⟩
| ⟨@routine⟩ ⟨Stmt⟩
| ⟨@safe⟩ *JuliaExpr*
| ⟨CallStmt⟩
⟨BlockStmt⟩ ::= *begin* ⟨Stmts⟩ *end*
⟨RevCond⟩ ::= *(JuliaExpr , JuliaExpr)*
⟨IfStmt⟩ ::= *if* ⟨RevCond⟩ ⟨Stmts⟩ [*else* ⟨Stmts⟩] *end*
⟨WhileStmt⟩ ::= *while* ⟨RevCond⟩ ⟨Stmts⟩ *end*
⟨Range⟩ ::= *JuliaExpr : JuliaExpr* [*: JuliaExpr*]
⟨ForStmt⟩ ::= *for ident* = ⟨Range⟩ ⟨Stmts⟩ *end*
⟨KwArg⟩ ::= *ident* = *JuliaExpr*
⟨KwArgs⟩ ::= [⟨KwArgs⟩ ,] ⟨KwArg⟩
⟨CallStmt⟩ ::= *JuliaExpr (* [⟨DataViews⟩] [*;* ⟨KwArgs⟩] *)*
⟨Constant⟩ ::= *num* | π
⟨InstrBinOp⟩ ::= += | -= | $\veebar$=
⟨InstrTrailer⟩ ::= [.] *(* [⟨DataViews⟩] *)*
⟨InstrStmt⟩ ::= ⟨DataView⟩ ⟨InstrBinOp⟩ *ident* [⟨InstrTrailer⟩]
⟨RevStmt⟩ ::= ~ ⟨Stmt⟩
⟨AncillaStmt⟩ ::= *ident* ← *JuliaExpr*
⟨TypecastStmt⟩ ::= *(JuliaExpr => JuliaExpr) (ident)*
⟨@routine⟩ ::= *@routine ident* ⟨Stmt⟩
⟨@safe⟩ ::= *@safe JuliaExpr*
⟨DataViews⟩ ::= ϵ
| ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩ ...
⟨DataView⟩ ::= ⟨DataView⟩ *[JuliaExpr]*
| ⟨DataView⟩ . *ident*
| *JuliaExpr (* ⟨DataView⟩ *)*
| ⟨DataView⟩ ′
| - ⟨DataView⟩
| ⟨Constant⟩
| *ident*

Appendix B: Instructions and Backward Rules

The translation of instructions to Julia functions
The list of instructions implemented in NiLang

instruction	translated	symbol
y += $f(args...)$	`PlusEq(f)(args...)`	$\oplus$
y −= $f(args...)$	`MinusEq(f)(args...)`	$\ominus$
$y \veebar= f(args...)$	`XorEq(f)(args...)`	$\odot$

Table III: Instructions and their compilation in NiLang.

instruction	output
SWAP(a, b)	b, a
ROT(a, b, θ)	$a\cos\theta - b\sin\theta, b\cos\theta + a\sin\theta, \theta$
IROT(a, b, θ)	$a\cos\theta + b\sin\theta, b\cos\theta - a\sin\theta, \theta$
y += a^b	$y + a^b, a, b$
y += exp(x)	$y + e^x, x$
y += log(x)	$y + \log x, x$
y += sin(x)	$y + \sin x, x$
y += cos(x)	$y + \cos x, x$
y += abs(x)	$y + \|x\|, x$
NEG(y)	$-y$
CONJ(y)	y'

Table IV: A collection of reversible instructions, "." is the broadcasting operations in Julia.

1. Backward rules for instructions

For function $\vec{y} = f(\vec{x})$, its Jacobian is $J_{ij} = \frac{\partial y_i}{\partial x_j}$ and its Hessian is $H^k_{ij} = \frac{\partial y_k}{x_i x_j}$. We have the following local Jacobians and Hessians on the above instructions.

1. a += b

$$J = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$
$$H = \mathbf{0}$$

The inverse is a −= b, its Jacobian is the inverse of the matrix above

$$J(f^{-1}) = J^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

In the following, we omit the Jacobians and Hessians of inverse functions.

2. a += $b * c$

$$J = \begin{pmatrix} 1 & c & b \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{bc} = H^a_{cb} = 1, else\ 0$$

3. $a \mathrel{+}= b/c$

$$J = \begin{pmatrix} 1 & 1/c & -b/c^2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{cc} = 2b/c^3,$$
$$H^a_{bc} = H^a_{cb} = -1/c^2, else\ 0$$

4. $a \mathrel{+}= b^c$

$$J = \begin{pmatrix} 1 & cb^{c-1} & b^c \log b \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{bc} = H^a_{cb} = b^{c-1} + cb^{c-1} \log b,$$
$$H^a_{bb} = (c-1)cb^{c-2},$$
$$H^a_{cc} = b^c \log^2 b, else\ 0$$

5. $a \mathrel{+}= e^b$

$$J = \begin{pmatrix} 1 & e^b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = e^b, else\ 0$$

6. $a \mathrel{+}= \log b$

$$J = \begin{pmatrix} 1 & 1/b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = -1/b^2, else\ 0$$

7. $a \mathrel{+}= \sin b$

$$J = \begin{pmatrix} 1 & \cos b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = -\sin b, else\ 0$$

8. $a \mathrel{+}= \cos b$

$$J = \begin{pmatrix} 1 & -\sin b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = -\cos b, else\ 0$$

9. $a \mathrel{+}= |b|$

$$J = \begin{pmatrix} 1 & \mathrm{sign}(b) \\ 0 & 1 \end{pmatrix}$$
$$H = \mathbf{0}$$

10. $a = -a$

$$J = \begin{pmatrix} -1 \end{pmatrix}$$
$$H = \mathbf{0}$$

11. $\mathrm{SWAP}(a, b) = (b, a)$

$$J = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$H = \mathbf{0}$$

12.

$$\mathrm{ROT}(a, b, \theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

$$J = \begin{pmatrix} \cos\theta & -\sin\theta & -b\cos\theta - a\sin\theta \\ \sin\theta & \cos\theta & a\cos\theta - b\sin\theta \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{a\theta} = H^a_{\theta,a} = -\sin\theta,$$
$$H^a_{b\theta} = H^a_{\theta,b} = -\cos\theta,$$
$$H^a_{\theta\theta} = -a\cos\theta + b\sin\theta,$$
$$H^b_{a\theta} = H^b_{\theta a} = \cos\theta,$$
$$H^b_{b\theta} = H^b_{\theta b} = -\sin\theta,$$
$$H^b_{\theta\theta} = -b\cos\theta - a\sin\theta, else\ 0$$

Appendix C: Learn by consistency

Consider a training that with input $\mathbf{x}^*$ and output $\mathbf{y}^*$, find a set of parameters $\mathbf{p}_x$ that satisfy $\mathbf{y}^* = f(\mathbf{x}^*, \mathbf{p}_x)$. In traditional machine learning, we define a loss $\mathcal{L} = \mathrm{dist}(\mathbf{y}^*, f(\mathbf{x}^*, \mathbf{p}_x))$ and minimize it with gradient $\frac{\partial L}{\partial \mathbf{p}_x}$. This works only when the target function is locally differentiable.

Here we provide an alternative by making use of reversibility. We construct a reversible program $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$, where $\mathbf{p}_x$ and $\mathbf{p}_y$ are "parameter" spaces on the input side and output side. The algorithm can be summarized as

Algorithm 2: Learn by consistency

Result: $\mathbf{p}_x$
Initialize $\mathbf{x}$ to $\mathbf{x}^*$, parameter space $\mathbf{p}_x$ to random.
if $\mathbf{p}_y$ *is* `null` **then**
  $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}^*)$
else
  $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$
  while $\mathbf{y} \not\approx \mathbf{y}^*$ **do**
    $\mathbf{y} = \mathbf{y}^*$
    $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}, \mathbf{p}_y)$.
    $\mathbf{x} = \mathbf{x}^*$
    $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$

Here, `parameter`$(\cdot)$ is a function for taking the parameter space. This algorithm utilizes the self-consistency relation

$$\mathbf{p}_x^* = \texttt{parameter}(f_r^{-1}(\mathbf{y}^*, \texttt{parameter}(f_r(\mathbf{x}^*, \mathbf{p}_x^*)))), \quad \text{(C1)}$$

Similar idea of training by consistency is used in self-consistent meanfield theory [74] in physics. Finding the self-consistent relation is crucial to a self-consistency based training. Here, the reversibility provides a natural self-consistency relation. However, it is not a silver bullet, let's consider the following example

```
@i function f1(y!, x, p!)
    p! += identity(x)
    y! -= exp(x)
    y! += exp(p!)
end

@i function f2(y!, x!, p!)
    p! += identity(x!)
    y! -= exp(x!)
    x! -= log(-y!)
    y! += exp(p!)
end

function train(f)
    loss = Float64[]
    p = 1.6
    for i=1:100
        y!, x = 0.0, 0.3
        @instr f(y!, x, p)
        push!(loss, y!)
        y! = 1.0
        @instr (~f)(y!, x, p)
    end
    loss
end
```

Functions `f1` and `f2` computes $f(x,p) = e^{(p+x)} - e^x$ and stores the output in a new memory `y!`. The only difference is `f2` uncomputes x arithmetically. The task of training is to find a p that make the output value equal to target value 1. After 100 steps, `f2` runs into the fixed point with x equal to 1 upto machine precision. However, parameters in `f1` does change at all. The training of `f1` fails because this function actually computes $\texttt{f1}(y,x,p) = y + e^{(p+x)} - e^x, x, x+p$, where the training parameter p is completely determined by the parameter space on the output side $x \cup x + p$. As a result, shifting y directly is the only approach to satisfy the consistency relation. On the other side, $\texttt{f2}(y,x,p) = y + e^{(p+x)} - e^x, \tilde{0}, x+p$, the output parameters $\tilde{0} \cup x + p$ can not uniquely determine input parameters p and x. Here, we use $\tilde{0}$ to denote the zero with rounding error.

By viewing $\mathbf{x}$ and parameters in $\mathbf{p}_x$ as variables, we can study the trainability from the information perspective.

***Theorem* 2.** *Only if the the conditional entropy $S(\mathbf{y}|\mathbf{p}_y)$ is nonzero, algorithm 2 is trainable.*

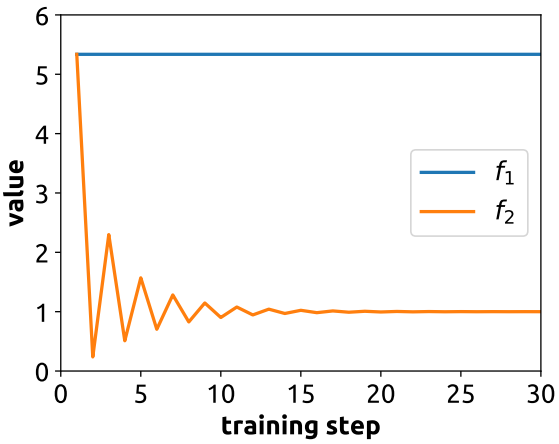


Figure 7: The output value `y!` as a function of self-consistent training step.

Proof. The above example reveals a fact that training impossible when output parameters completely determines input parameters (or $S(\mathbf{p}_x|\mathbf{p}_y) = 0$).

$$\begin{aligned} S(\mathbf{p}_x|\mathbf{p}_y) &= S(\mathbf{p}_x \cup \mathbf{p}_y) - S(\mathbf{p}_y) \\ &\le S\left((\mathbf{p}_x \cup \mathbf{x}) \cup \mathbf{p}_y\right) - S(\mathbf{p}_y), \\ &\le S\left((\mathbf{p}_y \cup \mathbf{y}) \cup \mathbf{p}_y\right) - S(\mathbf{p}_y), \\ &\le S(\mathbf{y}|\mathbf{p}_y). \end{aligned} \quad \text{(C2)}$$

The third line uses the bijectivity $S(\mathbf{x} \cup \mathbf{p}_x) = S(\mathbf{y} \cup \mathbf{p}_y)$. This inequality shows that when the parameter space on the output side satisfies $S(\mathbf{y}|\mathbf{p}_y) = 0$, i.e. contains all information to determine the output field, the input parameters are also completely determined by this parameter space, hence training can not work. □

In the above example, it corresponds to the case $S\left(e^{(x+y)-e^x}|x \cup x+y\right) = 0$ in $f1$. The solution is to remove the information redundancy in output parameter space through uncomputing as shown in $f2$.

Appendix D: Functions used in the main text

```
"""
    dot(out!, v1, v2)

dot product.
"""
@i function dot(out!, v1::Vector{T}, v2) where T
    for i = 1:length(v1)
        out! += v1[i]'*v2[i]
    end
end

"""
    norm2(out!, vec)

squared norm.
"""
@i function norm2(out!, vec::Vector{T}) where T
    anc1 ← zero(T)
    for i = 1:length(vec)
        anc1 += identity(vec[i]')
        out! += anc1*vec[i]
        anc1 -= identity(vec[i]')
    end
end
```

In `norm2`, we copied `vec[i]'` to `anc1` to avoid the same variable appear twice in the argument list of $\oplus(*)$, where the prime represents the adjoint dataview.

```
"""
    var_and_mean_sq(var!, mean!, sqv)

Variance and mean value from squared values `sqv`.
"""
@i function var_and_mean_sq(var!, mean!, sqv)
    sqmean ← zero(mean!)
    @inbounds for i=1:length(sqv)
        mean! += sqv[i] ^ 0.5
        var! += identity(sqv[i])
    end
    DIVINT(mean!, length(sqv))
    DIVINT(var!, length(sqv))
    sqmean += mean! ^ 2
    var! -= identity(sqmean)
    sqmean -= mean! ^ 2
    mulint(var!, length(sqv))
    divint(var!, length(sqv)-1)
end
```

Appendix E: CUDA compitibility

CUDA programming is playing a more and more important role in high performance computing. In Julia, one can write kernel functions in native Julia language with CUDAnative [75]. NiLang is compatible with CUDAnative, one can write a reversible kernel like the following.

```
using CuArrays, CUDAnative, GPUArrays
using NiLang, NiLang.AD

@i @inline function swap_kernel(state, mask1, mask2)
    @invcheckoff b ← (blockIdx().x-1) *
        blockDim().x + threadIdx().x - 1
    @invcheckoff if (b < length(state), ~)
        if (b&mask1==0 && b&mask2==mask2, ~)
            SWAP(state[b+1], state[b ⊻
                (mask1|mask2) + 1])
        end
    end
end
```

This kernel function simulates the SWAP gate in quantum computing. Here, we use the macro `@invcheckoff` to remove the reversibility checks. It is necessary because the possible error thrown in a kernel function can not be handled properly on a CUDA thread. One can launch this kernel function to GPUs with a single macro `@cuda` as show in the following using case.

```
julia> @i function instruct!(state::CuVector,
            gate::Val{:SWAP}, locs::Tuple{Int,Int})
           mask1 ← 1 << (tget(locs, 1)-1)
           mask2 ← 1 << (tget(locs, 2)-1)
           XY ← GPUArrays.thread_blocks_heuristic(
                length(state))
           @cuda threads=tget(XY,1) blocks=tget(XY,
                2) swap_kernel(state, mask1, mask2)
       end

julia> instruct!(CuArray(randn(8)),
            Val(:SWAP), (1,3))[1]
8-element CuArray{Float64,1,Nothing}:
 -0.06956048379200473
 -0.6464176838567472
 -0.06523362834285944
 -0.7314356941903547
  1.512329204247244
  0.9773772766637732
  1.6473223915215722
 -1.0631789613639087
```

Appendix F: Computing Fibonacci Numbers

The following is an example that everyone likes, computing Fibonacci number recursively.

```
using NiLang

@i function rfib(out!, n::T) where T
    n1 ← zero(T)
    n2 ← zero(T)
    @routine begin
        n1 += identity(n)
        n1 -= identity(1)
        n2 += identity(n)
        n2 -= identity(2)
    end
    if (value(n) <= 2, ~)
        out! += identity(1)
    else
        rfib(out!, n1)
        rfib(out!, n2)
    end
    ~@routine
end
```

The time complexity of this recursive algorithm is exponential to input `n`. It is also possible to write a reversible linear time for loop algorithm. A slightly non-trivial task is computing the first Fibonacci number that greater or equal to a certain number z, where a `while` statement is required.

```
@i function rfibn(n!, z)
    @safe @assert n! == 0
    out ← 0
    rfib(out, n!)
    while (out < z, n! != 0)
        ~rfib(out, n!)
        n! += identity(1)
        rfib(out, n!)
    end
    ~rfib(out, n!)
end
```

In this example, the postcondition `n!=0` in the `while` statement is false before entering the loop, and becomes true in later iterations. In the reverse program, the `while` statement stops at `n==0`. If executed correctly, a user will see the following result.

```
julia> rfib(0, 10)
(55, 10)

julia> rfibn(0, 100)
(12, 100)

julia> (~rfibn)(rfibn(0, 100)...)
(0, 100)
```

This examplifies how an addition postcondition provided by user can help reversing a control flow without caching controls.

Appendix G: Alternative approaches to Obtain Hessian

Here, `ibesselj'` is a callable instance of type `Grad{typeof(ibesselj)}`. The first parameters `Val(1)` specifies the position of loss in argument list. This function itself is reversible and differentiable, hence one can back-propagate this function to obtain Hessians as introduced in Sec. III B 2. In NiLang, it is implemented as `hessian_backback`.

```
julia>
    hessian_backback(ibesselj, (out!, 2, x); iloss=1)
3×3 Array{Float64,2}:
 0.0  0.0   0.0
 0.0  0.0   0.0
 0.0  0.0  -0.27505
```

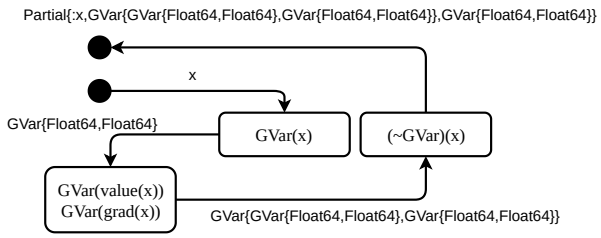


Figure 8: Data flow in obtaining second-order gradients with the recursive differentiation approach. Annotations on lines are data types used in the computation.

Fig. 8 shows the data flow in the four passes of computing Hessian. The first two passes obtains the gradients. Before entering the third pass, the program wraps each field in `GVar` with another layer of `GVar`. Then we pick a variable x_i and add 1 to the gradient field of its gradient `grad(grad(`x_i`))` in order to compute the i-th row of Hessian. Before entering the final pass, the `~GVar` is called. We can not unwrap `GVar` directly because although the values of gradients have been uncomputed to zero, the gradient fields of gradients may be nonzero. Instead, we use `Partial{:x}(obj)` to take field `x` of an object without erasing memory. By repeating the above procedure for different x_i, one can obtains the full Hessian matrix.

To obtain Hessians, we can also use the Hessian propagation approach as introduced in Sec. III B 3.

```
julia> ibesselj''(out!, 2, x; iloss=1)
(Loss(BeijingRing{Float64}(0.0, 1.0, 1)), 2,
BeijingRing{Float64}(3.0, 0.014998134978750133, 2))

julia> collect_hessian()
2×2 Array{Float64,2}:
 0.0   0.0
 0.0  -0.27505
```

`ibesselj''` computes the second-order gradients. It wraps variables with type `BeijingRing` [76] in the backward pass. `BeijingRing` records Jacobians and Hessians for a variable, where Hessians are stored in a global storage. Whenever an n-th variable or ancilla is created, we push a ring of size $2n-1$ to a global tape. Whenever an ancilla is deallocated, we pop a ring from the top. The n-th ring stores Hessian elements $H_{i\leq n,n}$ and $H_{n,i<n}$.The final result can be collected by calling `collect_hessian()`, which will read out the Hessian matrix that stored in the global storage.