

---

# Differentiate Everything with a Reversible Domain-Specific Language: supplementary materials

---

Anonymous Author(s)

Affiliation

Address

email

1 In Sec. 1.1, we introduce the detailed design of NiLang, including the grammar, the instruction set  
2 and the constructor. In Sec. 2, we show the example codes used in the benchmarks. In Sec. 3, we  
3 discuss some important issues, the space time tradeoff, the instruction set, the gradient on ancilla  
4 problem, the shared memory problem, and also the future directions to go.

## 5 1 NiLang in Detail

### 6 1.1 NiLang Grammar

7 To define a reversible function one can use “@i” plus a standard function definition like bellow

```
8  """  
  docstring...  
  """  
  @i function f(args..., kwargs...) where {...}  
    <stmts>  
  end
```

where the

9 definition of “<stmts>” are shown in the grammar page bellow. The following is a list of  
10 terminologies used in the definition of grammar

- 11 • *ident*, symbols
- 12 • *num*, numbers
- 13 •  $\epsilon$ , empty statement
- 14 • *JuliaExpr*, native Julia expression
- 15 •  $[ ]$ , zero or one repetitions.

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

18	$\langle \text{Stmts} \rangle$	$::=$	$\epsilon$   $\langle \text{Stmt} \rangle$   $\langle \text{Stmts} \rangle \langle \text{Stmt} \rangle$
	$\langle \text{Stmt} \rangle$	$::=$	$\langle \text{BlockStmt} \rangle$   $\langle \text{IfStmt} \rangle$   $\langle \text{WhileStmt} \rangle$   $\langle \text{ForStmt} \rangle$   $\langle \text{InstrStmt} \rangle$   $\langle \text{RevStmt} \rangle$   $\langle \text{AncillaStmt} \rangle$   $\langle \text{TypecastStmt} \rangle$   $\langle @routine \rangle \langle \text{Stmt} \rangle$   $\langle @safe \rangle \text{JuliaExpr}$   $\langle \text{CallStmt} \rangle$
	$\langle \text{BlockStmt} \rangle$	$::=$	<i>begin</i> $\langle \text{Stmts} \rangle$ <i>end</i>
	$\langle \text{RevCond} \rangle$	$::=$	( <i>JuliaExpr</i> , <i>JuliaExpr</i> )
	$\langle \text{IfStmt} \rangle$	$::=$	<i>if</i> $\langle \text{RevCond} \rangle \langle \text{Stmts} \rangle$ [ <i>else</i> $\langle \text{Stmts} \rangle$ ] <i>end</i>
	$\langle \text{WhileStmt} \rangle$	$::=$	<i>while</i> $\langle \text{RevCond} \rangle \langle \text{Stmts} \rangle$ <i>end</i>
	$\langle \text{Range} \rangle$	$::=$	<i>JuliaExpr</i> : <i>JuliaExpr</i> [ <i>:</i> <i>JuliaExpr</i> ]
	$\langle \text{ForStmt} \rangle$	$::=$	<i>for ident =</i> $\langle \text{Range} \rangle \langle \text{Stmts} \rangle$ <i>end</i>
	$\langle \text{KwArg} \rangle$	$::=$	<i>ident = JuliaExpr</i>
	$\langle \text{KwArgs} \rangle$	$::=$	[ $\langle \text{KwArgs} \rangle$ ,] $\langle \text{KwArg} \rangle$
	$\langle \text{CallStmt} \rangle$	$::=$	<i>JuliaExpr</i> ( [ $\langle \text{DataViews} \rangle$ ] [ <i>:</i> $\langle \text{KwArgs} \rangle$ ] )
	$\langle \text{Constant} \rangle$	$::=$	<i>num</i>   $\pi$   <i>true</i>   <i>false</i>
	$\langle \text{InstrBinOp} \rangle$	$::=$	<i>+=</i>   <i>-=</i>   $\forall=$
	$\langle \text{InstrTrailer} \rangle$	$::=$	[.] ( [ $\langle \text{DataViews} \rangle$ ] )
	$\langle \text{InstrStmt} \rangle$	$::=$	$\langle \text{DataView} \rangle \langle \text{InstrBinOp} \rangle$ <i>ident</i> [ $\langle \text{InstrTrailer} \rangle$ ]
	$\langle \text{RevStmt} \rangle$	$::=$	$\sim \langle \text{Stmt} \rangle$
	$\langle \text{AncillaStmt} \rangle$	$::=$	<i>ident</i> $\leftarrow$ <i>JuliaExpr</i>   <i>ident</i> $\rightarrow$ <i>JuliaExpr</i>
	$\langle \text{TypecastStmt} \rangle$	$::=$	( <i>JuliaExpr</i> $\Rightarrow$ <i>JuliaExpr</i> ) ( <i>ident</i> )
	$\langle @routine \rangle$	$::=$	<i>@routine ident</i> $\langle \text{Stmt} \rangle$
	$\langle @safe \rangle$	$::=$	<i>@safe JuliaExpr</i>
	$\langle \text{DataViews} \rangle$	$::=$	$\epsilon$   $\langle \text{DataView} \rangle$   $\langle \text{DataViews} \rangle$ , $\langle \text{DataView} \rangle$   $\langle \text{DataViews} \rangle$ , $\langle \text{DataView} \rangle$ ...
	$\langle \text{DataView} \rangle$	$::=$	$\langle \text{DataView} \rangle$ [ <i>JuliaExpr</i> ]   $\langle \text{DataView} \rangle$ . <i>ident</i>   <i>JuliaExpr</i> ( $\langle \text{DataView} \rangle$ )   $\langle \text{DataView} \rangle$ '   - $\langle \text{DataView} \rangle$   $\langle \text{Constant} \rangle$   <i>ident</i>

19 Table 1 shows the meaning of some selected statements and how they are reversed.

Statement	Meaning	Inverse
$\langle f \rangle(\langle \text{args} \rangle \dots)$	function call	$(\sim \langle f \rangle)(\langle \text{args} \rangle \dots)$
$\langle f \rangle.(\langle \text{args} \rangle \dots)$	broadcast a function call	$\langle f \rangle.(\langle \text{args} \rangle \dots)$
$\langle y \rangle += \langle f \rangle(\langle \text{args} \rangle \dots)$	inplace add instruction	$\langle y \rangle -= \langle f \rangle(\langle \text{args} \rangle \dots)$
$\langle y \rangle \vee = \langle f \rangle(\langle \text{args} \rangle \dots)$	inplace XOR instruction	$\langle y \rangle \vee = \langle f \rangle(\langle \text{args} \rangle \dots)$
$\langle a \rangle \leftarrow \langle \text{expr} \rangle$	allocate a new variable	$\langle a \rangle \rightarrow \langle \text{expr} \rangle$
begin $\langle \text{stmts} \rangle$ end	statement block	begin $\sim(\langle \text{stmts} \rangle)$ end
if ( $\langle \text{pre} \rangle$ , $\langle \text{post} \rangle$ ) $\langle \text{stmts1} \rangle$ else $\langle \text{stmts2} \rangle$ end	if statement	if ( $\langle \text{post} \rangle$ , $\langle \text{pre} \rangle$ ) $\sim(\langle \text{stmts1} \rangle)$ else $\sim(\langle \text{stmts2} \rangle)$ end
while ( $\langle \text{pre} \rangle$ , $\langle \text{post} \rangle$ ) $\langle \text{stmts} \rangle$ end	while statement	while ( $\langle \text{post} \rangle$ , $\langle \text{pre} \rangle$ ) $\sim(\langle \text{stmts} \rangle)$ end
for $\langle i \rangle = \langle m \rangle : \langle s \rangle : \langle n \rangle$ $\langle \text{stmts} \rangle$ end	for statement	for $\langle i \rangle = \langle m \rangle : -\langle s \rangle : \langle n \rangle$ $\sim(\langle \text{stmts} \rangle)$ end

Table 1: Basic statements in NiLang IR. “ $\sim$ ” is the symbol for reversing a statement or a function. “.” is the symbol for the broadcasting magic in Julia,  $\langle \text{pre} \rangle$  stands for precondition, and  $\langle \text{post} \rangle$  stands for postcondition “begin  $\langle \text{stmts} \rangle$  end” is the code block statement in Julia. It can be inverted by reversing the order as well as each element in it.

## 20 1.2 Instructions Used in Main Text

A table instructions used in the main text

instruction	output
SWAP( $a, b$ )	$b, a$
ROT( $a, b, \theta$ )	$a \cos \theta - b \sin \theta, b \cos \theta + a \sin \theta, \theta$
IROT( $a, b, \theta$ )	$a \cos \theta + b \sin \theta, b \cos \theta - a \sin \theta, \theta$
$y += a * b$	$y + a * b, a, b$
$y += a / b$	$y + a / b, a, b$
$y += a^b$	$y + a^b, a, b$
$y += \text{identity}(x)$	$y + x, x$
$y += \exp(x)$	$y + e^x, x$
$y += \log(x)$	$y + \log x, x$
$y += \sin(x)$	$y + \sin x, x$
$y += \cos(x)$	$y + \cos x, x$
$y += \text{abs}(x)$	$y +  x , x$
NEG( $y$ )	$-y$
INC( $y$ )	$y + 1$
DEC( $y$ )	$y - 1$

Table 2: Predefined reversible instructions in NiLang.

## 22 1.3 Reversible Constructors

23 So far, the language design is not too different from a traditional reversible language. To port  
 24 Julia’s type system better, we introduce dataviews. The type used in the reversible context is just a  
 25 standard Julia type with an additional requirement of having reversible constructors. The inverse of  
 26 a constructor is called a “destructor”, which unpacks data and deallocates derived fields. A  
 27 reversible constructor is implemented by reinterpreting the new function in Julia. Let us consider  
 28 the following statement.

```
29 x ← new{TX, TG}(x, g)
```

30 The above statement is similar to allocating an ancilla, except that it deallocates *g* directly at the  
 31 same time. Doing this is proper because *new* is special that its output keeps all information of its  
 32 arguments. All input variables that do not appear in the output can be discarded safely. Its inverse is

```
33 x → new{TX, TG}(x, g)
```

34 It unpacks structure *x* and assigns fields to corresponding variables in the argument list. The  
 35 following example shows a non-complete definition of 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
end

julia> GVar{Float64,Float64}(0.5, 0.0)
julia> (~GVar)(GVar{Float64,Float64}(0.5))
0.5
julia> (~GVar)(GVar{Float64,Float64}([0.5, 0.6]))
2-element Array{Float64,1}:
 0.5
 0.6

```

37 *GVar* has two fields that correspond to the value and gradient of a variable. Here, we put *@i* macro  
 38 before both *struct* and *function* statements. The ones before functions generate forward and  
 39 backward functions, while the one before *struct* moves *~GVar* functions to the outside of the type  
 40 definition. Otherwise, the inverse function will be ignored by Julia compiler.

41 Since an operation changes data inplace in *NiLang*, a field of an immutable instance should also be  
 42 “modifiable”. Let us first consider the following example.

```

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

```

44 In Julia language, the assign statement above will throw a syntax error because the function call “-”  
 45 can not be assigned, and GVar is an immutable type. In NiLang, we use the macro @assignback to  
 46 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},
47 3   chfield(arr[2].g, -, res[1]))
4 x = res[2]
5 y = res[3]
```

48 The first line `PlusEq{*})(-arr[2].g, x, y)` computes the output as a tuple of length 3. At  
 49 lines 2-3, `chfield(x, Val{:g}, val)` modifies the `g` field of `x` and `chfield(x, -, res[1])`  
 50 returns `-res[1]`. Here, modifying a field requires the default constructor of a type not overwritten.  
 51 The assignments in lines 4 and 5 are straightforward. We call a bijection of a field of an object a  
 52 “dataview” of this object, and it is directly modifiable in NiLang. The definition of dataview can be  
 53 found in Appendix 1.1.

## 54 2 Examples

55 In this section, we introduce several examples.

- 56 ○ sparse matrix dot product,
- 57 ○ first kind bessel function and memory oriented computational graph,
- 58 ○ solving the graph embedding problem.

59 All codes for this section and the next benchmark section are available in the [paper repository](#).

### 60 2.1 Sparse Matrices

61 Differentiating sparse matrices is useful in many applications, however, it can not benefit directly  
 62 from generic backward rules for the dense matrix because the generic rules do not keep the sparse  
 63 structure. In the following, we will show how to convert a irreversible Frobenius dot product code  
 64 to a reversible one to differentiate it. Here, the Frobenius dot product is defined as `trace(A'B)`. In  
 65 SparseArrays code base, it is implemented as follows.

```
function dot(A::AbstractSparseMatrixCSC{T1,S1},
  B::AbstractSparseMatrixCSC{T2,S2}
  ) where {T1,T2,S1,S2}
  m, n = size(A)
  size(B) == (m,n) || throw(DimensionMismatch("
66   matrices must have the same dimensions"))
  r = dot(zero(T1), zero(T2))
  @inbounds for j = 1:n
    ia = getcolptr(A)[j]
    ia_nxt = getcolptr(A)[j+1]
    ib = getcolptr(B)[j]
    ib_nxt = getcolptr(B)[j+1]
    if ia < ia_nxt && ib < ib_nxt
      ra = rowvals(A)[ia]
      rb = rowvals(B)[ib]
      while true
        if ra < rb
          ia += oneunit(S1)
          ia < ia_nxt || break
        elseif ra > rb
          ib += oneunit(S2)
          ib < ib_nxt || break
        else # ra == rb
          r += dot(nonzeros(A)[ia],
            nonzeros(B)[ib])
          ia += oneunit(S1)
          ib += oneunit(S2)
          ia < ia_nxt && ib < ib_nxt || break
          ra = rowvals(A)[ia]
          rb = rowvals(B)[ib]
        end
      end
    end
  end
  return r
end
```

67 It is easy to rewrite it in a reversible style with NiLang without sacrificing much performance.

```

68 @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
    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
    @invcheckoff branch_keeper → zeros(Bool, 2*m)
end

```

Here, all assignments are replaced with  $\leftarrow$  to indicate that the values of these variables must be returned at the end of this function scope. We put a “~” symbol in the postcondition field of if statements to indicate this postcondition is a dummy one that takes the same value as the precondition, i.e. the condition is not changed inside the loop body. If the precondition is changed by the loop body, one can use a `branch_keeper` vector to cache branch decisions. The value of `branch_keeper` can be restored through uncomputing (the “~” statement above). Finally, after checking the correctness of the program, one can turn off the reversibility checks by using the macro `@invcheckoff` macro to achieve better performance.

## 2.2 The first kind Bessel function

A Bessel function of the first kind of order  $\nu$  can be computed via Taylor expansion

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

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

```

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

```

This computational process could be diagrammatically represented as a computational graph as shown in Fig. 1 (a). The computational graph is a directed acyclic graph (DAG), where a node is a function and an edge is a data. An edge connects two nodes, one generates this data, and one consumes it. A computational graph is more likely a mathematical expression. It can not describe inplace functions and control flows conveniently because it does not have the notation for memory and loops.

Before showing the reversible implementation, we introduce how to obtain the product of a sequence of numbers reversibly. Consecutive multiplication requires an increasing size of tape to cache an intermediate state  $x_1 x_2 \dots x_n$ , since one can not deallocate the previous state  $x_1 x_2 \dots x_{n-1}$  directly since  $*$  and  $/$  are not considered as reversible here. To mitigate the space overhead, the standard approach in reversible computing is the pebble game model [Perumalla \(2013\)](#) (or the checkpointing

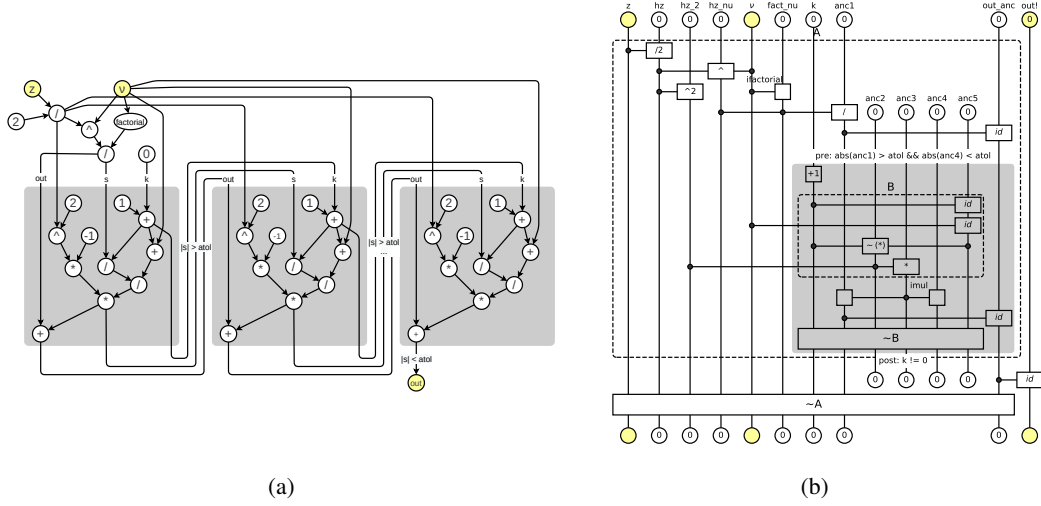


Figure 1: (a) The traditional computational graph for the irreversible implementation of the first kind Bessel function. A vertex (circle) is an operation, and a directed edge is a variable. The gray regions are the body of the unrolled while loop. (b) The memory oriented computational graph for the reversible implementation of the first kind Bessel function. Notations are explained in Fig. 3. The gray region is the body of a while loop. Its precondition and postcondition are positioned on the top and bottom, respectively.

technique in machine learning), where the cache size can be reduce in the cost of increasing the time complexity, however, constant cache size is not achievable in this scheme. Hence, 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, the third argument *anc!* is a *dirty ancilla*, which should be a value  $\approx 0$ . Line 2 computes the result and accumulates it to the dirty ancilla, so that we have an approximate output in *anc!*. Line 3 removes the content in *out!* approximately using the information stored in *anc!*. Line 4 swaps the contents in *out!* and *anc!*. Finally, we have an approximate output and a dirtier ancilla. The reason why this trick works here lies in the fact that  $\ast$  and  $/$  are mathematically reversible (except the zero point) to each other. One can approximately uncomputing the contents in the register at the cost of rounding errors. This rounding error introduced in such a way only affects the function output, and does not sacrifice the reversibility. With this approximate multiplier, we implement the reversible  $J_\nu$  as follows.

```

using NiLang, NiLang.AD

@i function ibesselj(out!, v, z; atol=1e-8)
    k ← 0
    fact_nu ← zero(v)
    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 ^ v
        halfz_power_2 += halfz ^ 2
        ifactorial(fact_nu, v)

        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(v)
        anc2 -= k * anc5
        anc3 += halfz_power_2 / anc2

    end
    imul(anc1, anc3, anc4)
    out_anc += identity(anc1)
    ~@routine
end
out! += identity(out_anc)
~@routine
end

@i function ifactorial(out!, n)
    anc ← zero(n)
    out! += identity(1)
    for i=1:n
        imul(out!, i, anc)
    end
end

@i @inline function imul(out!::T, x::T, anc!::T)
    where T<:Integer
    anc! += out! * x
    out! -= anc! ÷ x
    SWAP(out!, anc!)
end

```

107 The above algorithm uses a constant number of ancillas, while the time overhead is also a constant  
 108 factor. Ancilla anc4 plays the role of *dirty ancilla* in multiplication, and it is uncomputed rigorously  
 109 in the uncomputing stage marked by `~@routine`. This reversible program can be diagrammatically  
 110 represented as a memory oriented computational graph as shown in Fig. 1 (b). This diagram can  
 111 be used to analyze variables uncomputing. In this example, routine “B” uses `hz_2`, `v` and `k` as  
 112 control parameters, and changes the contents in `anc2`, `anc3` and `anc5`. The following `imul` and  
 113 `(:+=)` (`identity`) copies the result to output without changing these variables. Hence we can  
 114 apply the inverse routine `~B` to safely restore contents in `anc2`, `anc3` and `anc5`, and this exemplifies  
 115 the compute-copy-uncompute paradigm.

```

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

julia> Grad(ibesselj)(Val(1), out!, 2, x)
(Val{1}(), GVar(0.0, 1.0), 2, GVar(1.0, 0.2102436))

```

117 One can obtain gradients of this function by calling `Grad(ibesselj)`. Here, `Grad(ibesselj)`  
 118 returns a callable instance of type `Grad{typeof(ibesselj)}`. The first parameter `Val(1)`  
 119 specifies the position of loss in argument list. The Hessian can be obtained by feeding  
 120 dual-numbers into this gradient function.

```

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

julia> Grad(ibesselj)(Val(1), out!, 2, x)
(Val{1}(), GVar(0.0, 1.0), 2, GVar(1.0, 0.2102436))

julia> using ForwardDiff: Dual

julia> _, hxout!, _, hxx = Grad(ibesselj)(Val(1),
    Dual(out!, zero(out!)), 2, Dual(x, one(x)));

julia> grad(hxx).partials[1]
0.13446683844358093

```

122 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`  
 123 that store the Hessian  $\frac{\partial^2 \text{out!}}{\partial x^2}$ .



### 2.2.1 Benchmark

We differentiate the first type Bessel function in Appendix 2.2 and show the benchmarks in Table 3. In the table, 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  $\sim 2$  times slower than its irreversible counterpart. Since Bessel function has only one input argument, forward mode AD tools are faster than reverse mode AD, both source-to-source framework ForwardDiff and operator overloading framework Tapenade have the a comparable computing time with the pure function call.

	$T_{\min}/\text{ns}$	Space/KB
Julia-O	18	0
NiLang-O	32	0
Tapenade-O	32	0
ForwardDiff-G	38	0
NiLang-G	201	0
NiLang-G (CUDA)	1.4	0
ReverseDiff-G	1406	1.2
Zygote-G	22596	13.47
Tapenade-G (Forward)	30	0
Tapenade-G (Backward)	111	> 0

Table 3: Time and space used for computing objective (O) and gradient (G) of the first kind Bessel function  $J_2(1.0)$ . The CUDA benchmark time is averaged over a batch size of 4000, which is not a fair comparison but shows how much performance can we get from GPU in the parallel computing context.

NiLang.AD is the reverse mode AD submodule in NiLang, and it takes 11 times the native Julia program, and is also 2 times slower than Tapenade. However, the key point is, there is no extra memory allocation like stack operations in the whole computation. The controllable memory allocation of NiLang makes it compatible with CUDA program. In other backward mode AD like Zygote, ReverseDiff and Tapenade, the memory allocation in heap is nonzero due to the checkpointing.

### 2.3 Solving a graph embedding problem

Graph embedding can be used to find a proper representation for an order parameter Takahashi and Sandvik (2020) in condensed matter physics. Ref. Takahashi and Sandvik (2020) considers a problem of finding the minimum Euclidean space dimension  $k$  that a Petersen graph can embed into, that the distances between pairs of connected vertices are  $l_1$ , and the distance between pairs of disconnected vertices are  $l_2$ , where  $l_2 > l_1$ . The Petersen graph is shown in Fig. 2. Let us denote the set of connected and disconnected vertex pairs as  $L_1$  and  $L_2$ , respectively. This problem can be variationally solved with the following loss.

$$\begin{aligned} \mathcal{L} = & \text{Var}(\text{dist}(L_1)) + \text{Var}(\text{dist}(L_2)) \\ & + \exp(\text{relu}(\overline{\text{dist}(L_1)} - \overline{\text{dist}(L_2)} + 0.1))) - 1 \end{aligned} \quad (2)$$

The first line is a summation of distance variances in two sets of vertex pairs, where  $\text{Var}(X)$  is the variance of samples in  $X$ . The second line is used to guarantee  $l_2 > l_1$ , where  $\bar{X}$  means taking the average of samples in  $X$ . Its reversible implementation could be found in our benchmark repository.

We repeat the training for dimension  $k$  from 1 to 10. In each training, we fix two of the vertices and optimize the positions of the rest. Otherwise, the program will find the trivial solution with overlapped vertices. For  $k < 5$ , the loss is always much higher than 0, while for  $k \geq 5$ , we can get a loss close to machine precision with high probability. From the  $k = 5$  solution, it is easy to see  $l_2/l_1 = \sqrt{2}$ . An Adam optimizer with a learning rate 0.01 Kingma and Ba requires  $\sim 2000$  steps training.

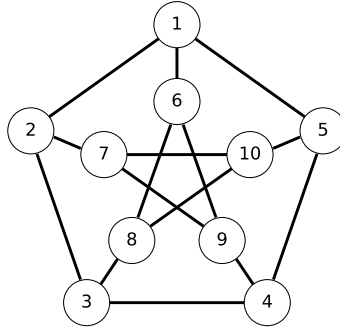


Figure 2: The Petersen graph has 10 vertices and 15 edges. We want to find a minimum embedding dimension for it.

154 The trust region Newton’s method converges much faster, which requires  $\sim 20$  computations of  
 155 Hessians to reach convergence. Although training time is comparable, the converged precision of  
 156 the later is much better.

## 157 2.4 Gaussian mixture model and bundle adjustment

158 You will find the source code in github repositories and  
 159 <https://github.com/JuliaReverse/NiGaussianMixture.jl>  
 160 <https://github.com/JuliaReverse/NiBundleAdjustment.jl>.

## 161 3 Discussion and outlooks

162 In this main text, we show how to realize a reversible programming eDSL and implement an  
 163 instruction level backward mode AD on top of it. It gives the user more flexibility to tradeoff  
 164 memory and computing time comparing with traditional checkpointing. The Julia implementation  
 165 NiLang gives the state-of-the-art performance and memory efficiency in obtaining first and  
 166 second-order gradients in applications, including first type Bessel function, sparse matrix  
 167 manipulations, solving graph embedding problem, Gaussian mixture model and bundle adjustment.  
 168 It provides the possibility to differentiate GPU kernels. In the following, we discuss some practical  
 169 issues about reversible programming, and several future directions to go.

### 170 3.1 Time Space Tradeoff

171 In history, there have been many discussions about time-space tradeoff on a reversible Turing  
 172 machine (RTM). In the most straightforward g-segment tradeoff scheme [Bennett \(1989\)](#); [Levine](#)  
 173 [and Sherman \(1990\)](#), an RTM model has either a space overhead that is proportional to computing  
 174 time  $T$  or a computational overhead that sometimes can be exponential to the program size  
 175 comparing with an irreversible counterpart. This result stops many people from taking reversible  
 176 computing seriously as a high-performance computing scheme. In the following, we try to explain  
 177 why the overhead of reversible computing is not as terrible as people thought.

178 First of all, the overhead of reversing a program is upper bounded by the checkpointing [Chen et al.](#)  
 179 [\(2016\)](#) strategy used in many traditional machine learning package that memorizes inputs of  
 180 primitives because checkpointing can be trivially implemented in reversible  
 181 programming. [Perumalla \(2013\)](#) Reversible programming provides some alternatives to reduce the  
 182 overhead. For example, accumulation is reversible, so that many BLAS functions can be  
 183 implemented reversibly without extra memory. Meanwhile, the memory allocation in some  
 184 iterative algorithms can often be reduced with the “arithmetic uncomputing” trick without  
 185 sacrificing reversibility, as shown in the `ibesselj` example in Appendix 2.2. Clever compiling  
 186 based on memory oriented computational graphs (Fig. 3 and Fig. 1 (b)) can also be used to help  
 187 user tradeoff between time and space. The overhead of a reversible program mainly comes from the

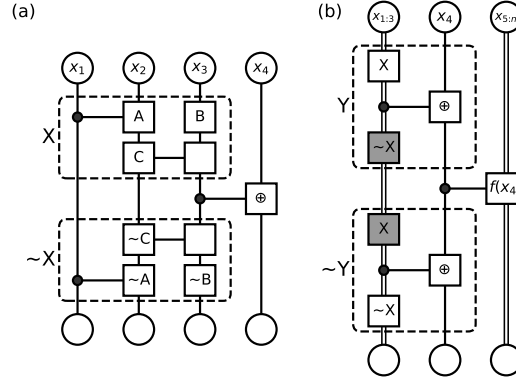


Figure 3: Two computational processes represented in memory oriented computational graph, where (a) is a subprogram in (b). In these graphs, a vertical single line represents one variable, a vertical double line represents multiple variables, and a parallel line represents a function. A dot at the cross represents a control parameter of a function and a box at the cross represents a mutable parameter of a function.

188 uncomputing of ancillas. It is possible to automatically uncompute ancillas by analyzing variable  
 189 dependency instead of asking users to write @routine and ~@routine pairs. In a hierarchical  
 190 design, uncomputing can appear in every memory deallocation (or symbol table reduction). To  
 191 quantify the overhead of uncomputing, we introduce the term uncomputing level as bellow.

192 **Definition 1** (uncomputing level). The log-ratio between the number of instructions of a reversible  
 193 program and its irreversible counterpart.

194 To explain how it works, we introduce the memory oriented computational graph, as shown in  
 195 Fig. 3. Notations are highly inspired by the quantum circuit representation. A vertical line is a  
 196 variable and a horizontal line is a function. When a variable is used by a function, depending on  
 197 whether its value is changed or not, we put a box or a dot at the line cross. It is different from  
 198 the computational graph for being a hypergraph rather than a simple graph, because a variable can  
 199 be used by multiple functions now. In panel (a). The subprogram in dashed box  $X$  is executed on  
 200 space  $x_{1:3}$  represents the computing stage. In the copying stage, the content in  $x_3$  is read out to a  
 201 pre-empted memory  $x_4$  through inplace add  $+=$ . Since this copy operation does not change contents  
 202 of  $x_{1:3}$ , we can use the uncomputing operation  $\sim X$  to undo all the changes to these registers. Now  
 203 we computing the result  $x_4$  without modifying the contents in  $x_{1:3}$ . If any of them is in a known  
 204 state, it can be deallocated immediately. In panel (b), we can use the subprogram defined in (a)  
 205 maked as  $Y$  to generate  $x_{5:n}$  without modifying the contents of variables  $x_{1:4}$ . It is easy to see that  
 206 although this uncompute-copy-uncompute design pattern can restore memories to known state, it  
 207 has computational overhead. Both  $X$  and  $\sim X$  are executed twice in the program (b), which is not  
 208 necessary. We can cancel a pair of  $X$  and  $\sim X$  (the gray boxes). By doing this, we are not allowed to  
 209 deallocate the memory  $x_{1:3}$  during computing  $f(x_{5:n})$ . This is the famous time-space tradeoff that  
 210 playing the central role in reversible programming.

211 From the lowest instruction level, whenever we reduce the symbol table (or space), the  
 212 computational cost doubles. The computational overhead grows exponentially as the  
 213 uncomputing level increases, which can be seen from some of the benchmarks in the main text.  
 214 In sparse matrix multiplication and dot product, we don't introduce uncomputing in the most time  
 215 consuming part, so it is  $\sim 0$ . The space overhead is  $2 \times m$  to keep the branch decisions, which is even  
 216 much smaller than the memory used to store row indices. in Gaussian mixture model, the most  
 217 time consuming matrix-vector multiplication is doubled, so it is  $\sim 1$ . The extra memory usage is  
 218 approximately 0.5% of the original program. In the first kind Bessel function and bundle  
 219 adjustment program, the most time consuming parts are (nestedly) uncomputed twice, hence their  
 220 uncomputing level is  $\sim 2$ . Such aggressive uncomputing makes zero memory allocation possible.

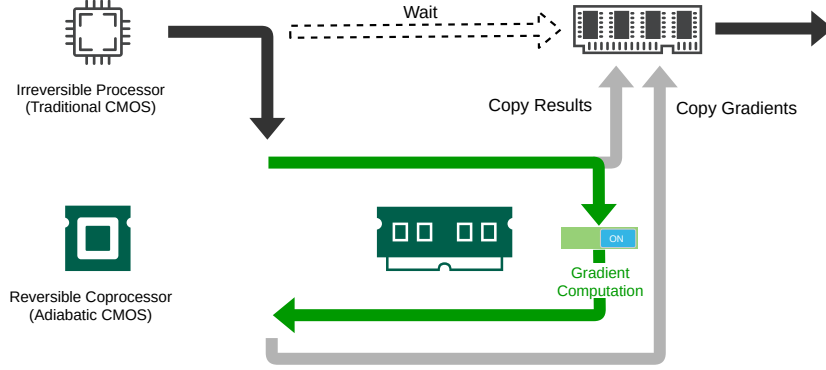


Figure 4: Energy efficient AI co-processor. Green arrows represents energy efficient operations on reversible devices.

### 3.2 Differentiability as a Hardware Feature

So far, our eDSL is compiled to Julia language. It relies on Julia’s multiple dispatch to differentiate a program, which requires users to write generic programs. A more liable AD should be a hardware or micro instruction level feature. In the future, we can expect NiLang being compiled to reversible instructions [Vieri \(1999\)](#) and executed on a reversible device. A reversible devices can play a role of differentiation engine as shown in the hetero-structural design in Fig. 4. It defines a reversible instruction set and has a switch that controls whether the instruction calls a normal instruction or an instruction that also updates gradients. When a program calls a reversible differentiable subroutine, the reversible co-processor first marches forward, compute the loss and copy the result to the main memory. Then the co-processor execute the program backward and uncall instructions, initialize and updating gradient fields at the same time. After reaching the starting point of the program, the gradients are transferred to the global memory. Running AD program on a reversible device can save energy. Theoretically, the reversible routines do not necessarily cost energy, the only energy bottleneck is copying gradient and outputs to the main memory.

### 3.3 The connection to Quantum programming

A Quantum device [Nielsen and Chuang \(2002\)](#) is a special reversible hardware that features quantum entanglement. The instruction set of classical reversible programming is a subset of quantum instruction set. However, building a universal quantum computer is difficult. Unlike a classical state, a quantum state can not be cloned. Meanwhile, it loses information by interacting with the environment. Classical reversible computing does not enjoy the quantum advantage, nor the quantum disadvantages of non-cloning and decoherence, but it is a model that we can try directly with our classical computer. It is technically 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 elementary components in reversible computing. For example, quantum Fourier transformation provides an alternative to the reversible adders and multipliers by introducing the Hadamard and CPHASE quantum gates [Ruiz-Perez and Garcia-Escartin \(2017\)](#). From the programming languages’s perspective, most quantum programming language preassumes the existence of a classical coprocessor to control quantum devices [Svore et al. \(2018\)](#). It is also interesting to know what is a native quantum control flow like, and does quantum entanglement provide speed up to automatic differentiation? We believe the reversible compiling technologies will open a door to study quantum compiling.

### 3.4 Gradient on ancilla problem

In this subsection, we introduce an easily overlooked problem in our reversible AD framework. An ancilla can sometimes carry a nonzero gradient when it is deallocated. As a result, the gradient program can be irreversible in the local scope. In NiLang, we drop the gradient field of ancillas instead of raising an error. In the following, we justify our decision by proving the following theorem.

257 **Theorem 1.** Deallocating an ancilla with constant value field and nonzero gradient field does not  
 258 introduce incorrect gradients.

259 *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  
 260 values of an ancilla. Since both  $a, b$  are constants that are independent of input  $\mathbf{x}^{i-1}$ , we have

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

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

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

264 where the second term with  $\frac{\partial \mathbf{x}^L}{\partial b}$  vanishes naturally. We emphasis here, the value part of discarded  
 265 ancilla must be a constant.  $\square$

### 266 3.5 Shared read and write problem

267 One should be careful about shared read in reversible programming AD, because the shared read  
 268 can introduce shared write in the adjoint program. Let's begin with the following expression.

```
269 y += x * y
```

270 Most people will agree that this statement is not reversible and should not be allowed because it  
 271 changes input variables. We call it the *simultaneous read-and-write* issue. However, the following  
 272 expression with two same inputs is a bit subtle.

```
273 y += x * x
```

274 It is reversible, but should not be allowed in an AD program because of the *shared write* issue. It  
 275 can be seen directly from the expanded expression.

```
276 julia> macroexpand(Main, :(@instr y += x * x))
quote
  var"##253" = ((PlusEq{*}) (y, x, x))
  begin
    y = (NiLangCore.wrap_tuple(var"##253"))[1]
    x = (NiLangCore.wrap_tuple(var"##253"))[2]
    x = (NiLangCore.wrap_tuple(var"##253"))[3]
  end
end
```

277 In an AD program, the gradient field of  $\mathbf{x}$  will be updated. The later assignment to  $x$  will overwrite  
 278 the former one and introduce an incorrect gradient. One can get free of this issue by avoiding using  
 279 same variable in a single instruction

```
280 anc ← zero(x)
    anc += identity(x)
    y += x * anc
    anc -= identity(x)
```

281 or equivalently,

```
282 y += x ^ 2
```

Share variables in an instruction can be easily identified and avoided. However, it becomes tricky when one runs the program in a parallel way. For example, in CUDA programming, every thread may write to the same gradient field of a shared scalar. How to solve the shared write in CUDA programming is still an open problem, which limits the power of reversible programming AD on GPU.

### 3.6 Several future directions

We can use NiLang to solve some existing issues related to AD. Reversible programming can make use of reversibility to save memory. Reversibility has been used in reducing the memory allocations in machine learning models such as recurrent neural networks [MacKay et al. \(2018\)](#), Hyperparameter learning [Maclaurin et al. \(2015\)](#) and residual neural networks [Behrmann et al. \(2018\)](#). We can use it to generate AD rules for existing machine learning packages like [ReverseDiff](#), Zygote [Innes et al. \(2019\)](#), Knet [Yuret \(2016\)](#), and Flux [Innes et al. \(2018\)](#). Many backward rules for sparse arrays and linear algebra operations have not been defined yet in these packages. We can also use the flexible time-space tradeoff in reversible programming to overcome the memory wall problem in some applications. A successful, related example is the memory-efficient domain-specific AD engine in quantum simulator Yao [Luo et al. \(2019\)](#). This domain-specific AD engine is written in a reversible style and solved the memory bottleneck in variational quantum simulations. It also gives so far the best performance in differentiating quantum circuit parameters. Similarly, we can write memory-efficient normalizing flow [Kobyzev et al. \(2019\)](#) in a reversible style. Normalizing flow is a successful class of generative models in both computer vision [Kingma and Dhariwal \(2018\)](#) and quantum physics [Dinh et al. \(2016\)](#); [Li and Wang \(2018\)](#), where its building block bijector is reversible. We can use a similar idea to differentiate reversible integrators [Hut et al. \(1995\)](#); [Laikov \(2018\)](#). With reversible integrators, it should be possible to rewrite the control system in robotics [Giftthaler et al. \(2017\)](#) in a reversible style, where scalar is a first-class citizen rather than tensor. Writing a reversible control program should boost training performance. Reversibility is also a valuable resource for training.

To solve the above problems better, reversible programming should be improved from multiple perspectives. First, we need a better compiler for compiling reversible programs. To be specific, a compiler that admits mutability of data, and handle shared read and write better. Then, we need a reversible number system and instruction set to avoid rounding errors and support reversible control flows better. There are proposals of reversible floating point adders and multipliers, however these designs require allocating garbage bits in each operation [Nachtigal et al. \(2010, 2011\)](#); [Nguyen and Meter \(2013\)](#); [Häner et al. \(2018\)](#). In NiLang, one can simulate rigorous reversible arithmetic with the fixed-point number package [FixedPointNumbers](#). However, a more efficient reversible design requires instruction-level support. Some other numbers systems are reversible under  $\ast =$  and  $/ =$  rather than  $+=$  and  $-=$ , including [LogarithmicNumbers](#) [Taylor et al. \(1988\)](#) and [TropicalNumbers](#). They are powerful tools to solve domain specific problems, for example, we have an upcoming work about differentiating over tropical numbers to solve the ground state configurations of a spinglass system efficiently. We also need `comefrom` like instruction as a partner of `goto` to specify the postconditions in our instruction set. Finally, although we introduced that the adiabatic CMOS as a better choice as the computing device in a spacecraft [DeBenedictis et al. \(2017\)](#). There are some challenges in the hardware side too, one can find a proper summary of these challenges in Ref. [Frank \(2005\)](#).

Solutions to these issues requires the participation of people from multiple fields.

### References

- K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).
- 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 \[cond-mat.str-el\]](#).
- D. P. Kingma and J. Ba, [arXiv:1412.6980](#).
- C. H. Bennett, *SIAM Journal on Computing* **18**, 766 (1989).
- R. Y. Levine and A. T. Sherman, *SIAM Journal on Computing* **19**, 673 (1990).



334 T. Chen, B. Xu, C. Zhang, and C. Guestrin, [CoRR abs/1604.06174](#) (2016), [arXiv:1604.06174](#) .

335 C. J. Vieri, *Reversible Computer Engineering and Architecture*, [Ph.D. thesis](#), Cambridge, MA, USA  
336 (1999), [aAI0800892](#).

337 M. A. Nielsen and I. Chuang, “Quantum computation and quantum information,” (2002).

338 L. Ruiz-Perez and J. C. Garcia-Escartin, [Quantum Information Processing](#) **16** (2017),  
339 [10.1007/s11128-017-1603-1](#).

340 K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov,  
341 M. Mykhailova, and A. Paz, [Proceedings of the Real World Domain Specific Languages](#)  
342 [Workshop 2018 on - RWDSL2018](#) (2018), [10.1145/3183895.3183901](#).

343 M. MacKay, P. Vicol, J. Ba, and R. B. Grosse, in *Advances in Neural Information Processing*  
344 *Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and  
345 R. Garnett (Curran Associates, Inc., 2018) pp. 9029–9040.

346 D. Maclaurin, D. Duvenaud, and R. Adams, in *Proceedings of the 32nd International Conference*  
347 *on Machine Learning*, Proceedings of Machine Learning Research, Vol. 37, edited by F. Bach and  
348 D. Blei (PMLR, Lille, France, 2015) pp. 2113–2122.

349 J. Behrmann, D. Duvenaud, and J. Jacobsen, [CoRR abs/1811.00995](#) (2018), [arXiv:1811.00995](#) .

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

352 D. Yuret (2016).

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

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

357 I. Kobyzev, S. Prince, and M. A. Brubaker, “Normalizing flows: An introduction and review of  
358 current methods,” (2019), [arXiv:1908.09257 \[stat.ML\]](#) .

359 D. P. Kingma and P. Dhariwal, in *Advances in Neural Information Processing Systems 31*, edited  
360 by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran  
361 Associates, Inc., 2018) pp. 10215–10224.

362 L. Dinh, J. Sohl-Dickstein, and S. Bengio, “Density estimation using real nvp,” (2016),  
363 [arXiv:1605.08803 \[cs.LG\]](#) .

364 S.-H. Li and L. Wang, [Physical Review Letters](#) **121** (2018), [10.1103/physrevlett.121.260601](#).

365 P. Hut, J. Makino, and S. McMillan, *The Astrophysical Journal* **443**, L93 (1995).

366 D. N. Laikov, [Theoretical Chemistry Accounts](#) **137** (2018), [10.1007/s00214-018-2344-7](#).

367 M. Gifftaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, [Advanced Robotics](#)  
368 **31**, 1225–1237 (2017).

369 M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on*  
370 *Nanotechnology* (2010) pp. 233–237.

371 M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on*  
372 *Nanotechnology* (2011) pp. 451–456.

373 T. D. Nguyen and R. V. Meter, “A space-efficient design for reversible floating point adder in  
374 quantum computing,” (2013), [arXiv:1306.3760 \[quant-ph\]](#) .

375 T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, “Quantum circuits for floating-point  
376 arithmetic,” (2018), [arXiv:1807.02023 \[quant-ph\]](#) .

- 377 F. J. Taylor, R. Gill, J. Joseph, and J. Radke, IEEE Transactions on Computers **37**, 190 (1988).
- 378 E. P. DeBenedictis, J. K. Mee, and M. P. Frank, Computer **50**, 76 (2017).
- 379 M. P. Frank, in *Proceedings of the 2nd Conference on Computing Frontiers* (2005) pp. 385–390.