Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Automatic Differentiation: Inverse Accumulation Mode

Barak A. Pearlmutter* Jeffrey Mark Siskind†

Abstract. We show that, under certain circumstances, it is possible to automatically compute Jacobian-inverse-vector and Jacobian-inverse-transpose-vector products about as efficiently as Jacobian-vector and Jacobian-transpose-vector products. The key insight is to notice that the Jacobian corresponding to the use of one basis function is of a form whose sparsity is invariant to inversion. The main restriction of the method is a constraint on the number of active variables, which suggests a variety of techniques or generalization to allow the constraint to be enforced or relaxed. This technique has the potential to allow the efficient direct calculation of Newton steps as well as other numeric calculations of interest.

1 Inverse automatic differentiation: the dream. Automatic Differentiation (AD) is the mechanical transformation of computer programs to calculate derivatives of interest, with useful complexity guarantees. The two most important "modes" of AD are forward and reverse, which access the Jacobian (the matrix of derivatives of each output of the computation with respect to each input) by multiplication, or transpose-multiplication, with a vector. Here we consider first-order numeric computations, where inputs and outputs are vectors of reals. Given the primal computation

$$y = f(x) \qquad \text{with} \qquad f : \mathbb{R}^m \to \mathbb{R}^n$$

and therefore $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, we use $\mathbf{J}_{f(x)} \in \mathbb{R}^{n\times m}$ for the Jacobian of the function f at x, whose $(i,j)^{\text{th}}$ element is $\partial f_i(x)/\partial x_j$. Forward and Reverse AD compute

$$\dot{y} = \mathbf{J}_{f(x)}\,\dot{x} \qquad \text{and} \qquad \bar{x} = \mathbf{J}^{\mathsf{T}}_{f(x)}\,\bar{y} \tag{1.1}$$

respectively, with $\dot{\ }$ and $\bar{\ }$ denoting tangents and cotangents. Our objective here is to solve for the starred vectors on the right-hand sides of

$$\bar{x}^* = \mathbf{J}^{\mathsf{T}}_{f(x)}\,\bar{y}^* \qquad \text{and} \qquad \dot{y}^* = \mathbf{J}_{f(x)}\,\dot{x}^*. \tag{1.2}$$

If this solution can be done efficiently, it would allow efficient Newton steps (where f is a gradient calculation, say) and other sorts of second-order optimization. For this problem to be well posed it is necessary for $\mathbf{J}_{f(x)}$ to be invertible, so $n = m$, and as we shall see, further restrictions on the form of f will be required. Inverse Jacobians can be used to find roots of systems of equations. Inverse Hessians (which can be computed with Inverse AD over traditional AD) can be used for second order optimization, and for this reason are the topic of intensive research in the optimization community [3].

*Department of Computer Science, Maynooth University, Maynooth, Co. Kildare, Ireland (http://barak.pearlmutter.net).
†Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907-2035, USA (http://www.ece.purdue.edu/~qobi/).

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

2 Inverse automatic differentiation: the reality. Let us review Forward and Reverse AD. Since we are evaluating the function f at a point x, we consider control flow resolved and represent the corresponding computation as a data flow graph: a DAG whose edges hold "active" reals and whose vertices represent numeric basis functions such as

$$+, -, \times, \div, \sqrt{\cdot}, \log, \exp, \sin, \cos, \tan^{-1}, \textit{etc.},$$

that are primitives, intrinsics, or library functions in a typical programming language. There are n edges entering from the inputs $x_1, \ldots, x_n$, and n edges exiting to the outputs $y_1, \ldots, y_n$. If we topologically sort the data flow graph, and cut it before and after each vertex, we observe that the primal computation proceeds through a sequence of $T+1$ machine states, $\mathbf{x}_0, \ldots, \mathbf{x}_T$ holding a vector of active variables, where the initial and final states are the input and output of the computation,

$$\mathbf{x}_0 = x \qquad \text{and} \qquad \mathbf{x}_T = y,$$

respectively. We use f_t for the transition function from the machine state at time $t-1$ to the machine state at time t,

$$\mathbf{x}_t = f_t(\mathbf{x}_{t-1}),$$

and $\mathbf{J}_t$ for the Jacobian of the function f_t at $\mathbf{x}_{t-1}$, keeping in mind that f_t involves applying a single numeric basis function to some elements of $\mathbf{x}_{t-1}$ and putting the result in some elements of $\mathbf{x}_t$, copying the other elements unchanged.

Since the function f is the decomposition of the transition functions,

$$f = f_T \circ f_{T-1} \circ \cdots \circ f_2 \circ f_1,$$

the Jacobian matrix is a product,

$$\mathbf{J}_{f(x)} = \mathbf{J}_T\, \mathbf{J}_{T-1} \cdots \mathbf{J}_2\, \mathbf{J}_1,$$

and Forward and Reverse AD amount to appropriate associativity:

$$\dot{y} = \mathbf{J}_{f(x)}\, \dot{x} = \mathbf{J}_T(\mathbf{J}_{T-1} \cdots (\mathbf{J}_2(\mathbf{J}_1\, \dot{x}))\cdots) \qquad \text{and} \qquad \bar{x} = \mathbf{J}_{f(x)}^{\top}\, \bar{y} = \mathbf{J}_1^{\top}(\mathbf{J}_2^{\top} \cdots (\mathbf{J}_{T-1}^{\top}(\mathbf{J}_T^{\top}\, \bar{y}))\cdots). \tag{2.1}$$

Solving (1.2) in the form of (2.1) while assuming each $\mathbf{J}_t$ is invertible is the basic idea of *Forward Inverse Accumulation* and *Reverse Inverse Accumulation:*

$$\bar{y}^* = \mathbf{J}_T^{-\top}(\mathbf{J}_{T-1}^{-\top} \cdots (\mathbf{J}_2^{-\top}(\mathbf{J}_1^{-\top}\, \bar{x}^*))\cdots) \qquad \text{and} \qquad \dot{x}^* = \mathbf{J}_1^{-1}(\mathbf{J}_2^{-1} \cdots (\mathbf{J}_{T-1}^{-1}(\mathbf{J}_T^{-1}\, \dot{y}^*))\cdots), \tag{2.2}$$

where $\mathbf{M}^{-\top} = (\mathbf{M}^{-1})^{\top}$. These will be practical if the matrix-vector products $\mathbf{J}_t^{-1}\dot{y}^*$ and $\mathbf{J}_t^{-\top}\bar{x}^*$ can be calculated efficiently.

Assuming the computation of f is constant-width, so $\mathbf{x}_t \in \mathbb{R}^n$, and its local linearization is invertible, then each f_t must write its result to a slot where one of the inputs to the invoked basis function was stored, yielding Jacobians of the form

$$\begin{array}{c} R_t \\ \downarrow \end{array}$$

$$R_t \rightarrow \begin{bmatrix} 1 &&&&&& \\ & \ddots &&&&& \\ && 1 &&&& \\ &&& a &&& \\ &&&& 1 && \\ &&&&& \ddots & \\ &&&&&& 1 \end{bmatrix} \tag{2.3a}$$

$$a = \frac{\partial g(\mathbf{x}_{t-1}[R_t])}{\partial \mathbf{x}_{t-1}[R_t]}$$

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

for unary basis functions g that read and write to variable/slot R_t, and

$$
\begin{array}{c}
 \quad \begin{array}{cccccccc} & & & R_t & S_t & & \\ & & & \downarrow & \downarrow & & \end{array} \\
R_t \rightarrow \left[\begin{array}{ccccccc}
1 & & & & & & \\
 & \ddots & & & & & \\
 & & 1 & & & & \\
 & & & a & b & & \\
 & & & & 1 & & \\
 & & & & & \ddots & \\
 & & & & & & 1
\end{array}\right]
\end{array} \tag{2.3b}
$$

$$a = \frac{\partial h(\mathbf{x}_{t-1}[R_t],\mathbf{x}_{t-1}[S_t])}{\partial \mathbf{x}_{t-1}[R_t]}$$

$$b = \frac{\partial h(\mathbf{x}_{t-1}[R_t],\mathbf{x}_{t-1}[S_t])}{\partial \mathbf{x}_{t-1}[S_t]}$$

for binary basis functions h that read from variables/slots R_t and S_t and write to variable/slot R_t. We now note that these Jacobians can be trivially inverted. If we consider only variables involved in the basis function being invoked, and reorder them so the output values are first, a basis function with k inputs and a scalar output results in

$$\mathbf{J}_t = \left(\begin{array}{c|ccc} a & b_1 & \cdots & b_{k-1} \\ \hline \mathbf{0} & & \mathbf{I} & \end{array}\right) \tag{2.4a}$$

and

$$\mathbf{J}_t^{-1} = \left(\begin{array}{c|ccc} \frac{1}{a} & -\frac{b_1}{a} & \cdots & -\frac{b_{k-1}}{a} \\ \hline \mathbf{0} & & \mathbf{I} & \end{array}\right). \tag{2.4b}$$

We can generalize from scalar to l outputs, giving the form

$$\mathbf{J}_t = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \tag{2.4c}$$

and

$$\mathbf{J}_t^{-1} = \begin{pmatrix} \mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}, \tag{2.4d}$$

where $\mathbf{A} : l \times l$ and $\mathbf{B} : l \times (k-l)$. Although $\mathbf{J}_t$ is not structurally symmetric, $\mathbf{J}_t^{-1}$ has the same structural sparsity as $\mathbf{J}_t$. And although the amount of arithmetic is the same as for conventional Forward and Reverse modes, these are transposed, so Forward Inverse Mode writes to the derivative-related quantities associated with *all* involved variables of each basis function invocation, while Reverse Inverse Mode writes only to the quantities associated with slots *written to* in the primal computation of each basis function.

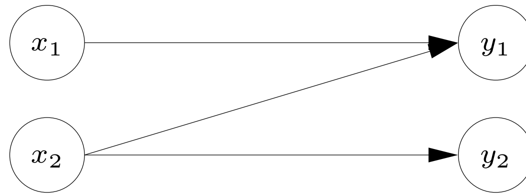
Figure 2.1 shows how all four AD modes transform atomic portions of a computation graph. Figure 2.2 illustrates all four AD modes on a simple program. Figure 2.3 illustrates how Figure 2.2(bcef) are derived from Figure 2.2(a) with the transformations of Figure 2.1 together with the layering technique of [15].

Traditional forward mode can be computed without saving intermediate values from the primal, as the primal and tangent can be computed in tandem. However, traditional reverse mode requires a tape to save the intermediate values from the primal computed during the forward sweep for use in reverse order during the reverse sweep. Analogously, forward inverse mode does not require a tape while reverse inverse mode does.

It is apparent from (2.4c) that any computation step that takes k inputs and produces l outputs will have an invertible Jacobian if $\mathbf{A} : l \times l$ is nonsingular. In particular, when $l = 1$, that will be when $a \neq 0$.

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

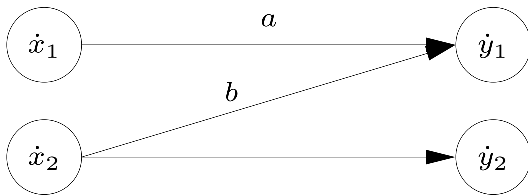
Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy



(a) Primal Code Step

$y_1 = f(x_1, x_2)$; $y_2 = x_2$

- fan in means multiple arguments

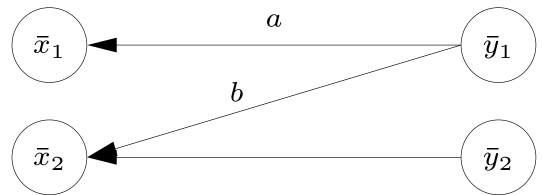


(b) Forward Transform Step: $\overrightarrow{\mathcal{J}}$

$a = \frac{\partial y_1}{\partial x_1}$; $b = \frac{\partial y_1}{\partial x_2}$

$\mathbf{J}_{\mathbf{f}_t(\mathbf{x}_{t-1})} = \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}$; $\dot{\mathbf{y}} = \mathbf{J}\dot{\mathbf{x}}$

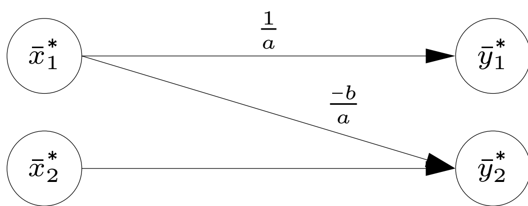
- same graph structure as primal
- edge labels denote multiplication
- missing edge labels are 1s
- fan in denotes addition



(c) Reverse Transform Step: $\overleftarrow{\mathcal{J}}$

$\mathbf{J}^{\mathsf{T}}_{\mathbf{f}_t(\mathbf{x}_{t-1})} = \begin{pmatrix} a & 0 \\ b & 1 \end{pmatrix}$; $\bar{\mathbf{x}} = \mathbf{J}^{\mathsf{T}}\bar{\mathbf{y}}$

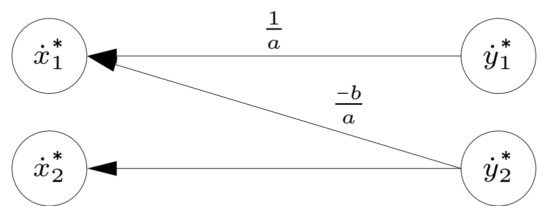
- edge reversal from forward mode



(d) Forward-Inverse Step: $\underrightarrow{\mathcal{L}}$

$\mathbf{J}^{-\mathsf{T}}_{\mathbf{f}_t(\mathbf{x}_{t-1})} = \begin{pmatrix} \frac{1}{a} & 0 \\ \frac{-b}{a} & 1 \end{pmatrix}$; $\bar{\mathbf{y}}^* = \mathbf{J}^{-\mathsf{T}}\bar{\mathbf{x}}^*$

- edge reversal from reverse-inverse



(e) Reverse-Inverse Step: $\underleftarrow{\mathcal{L}}$

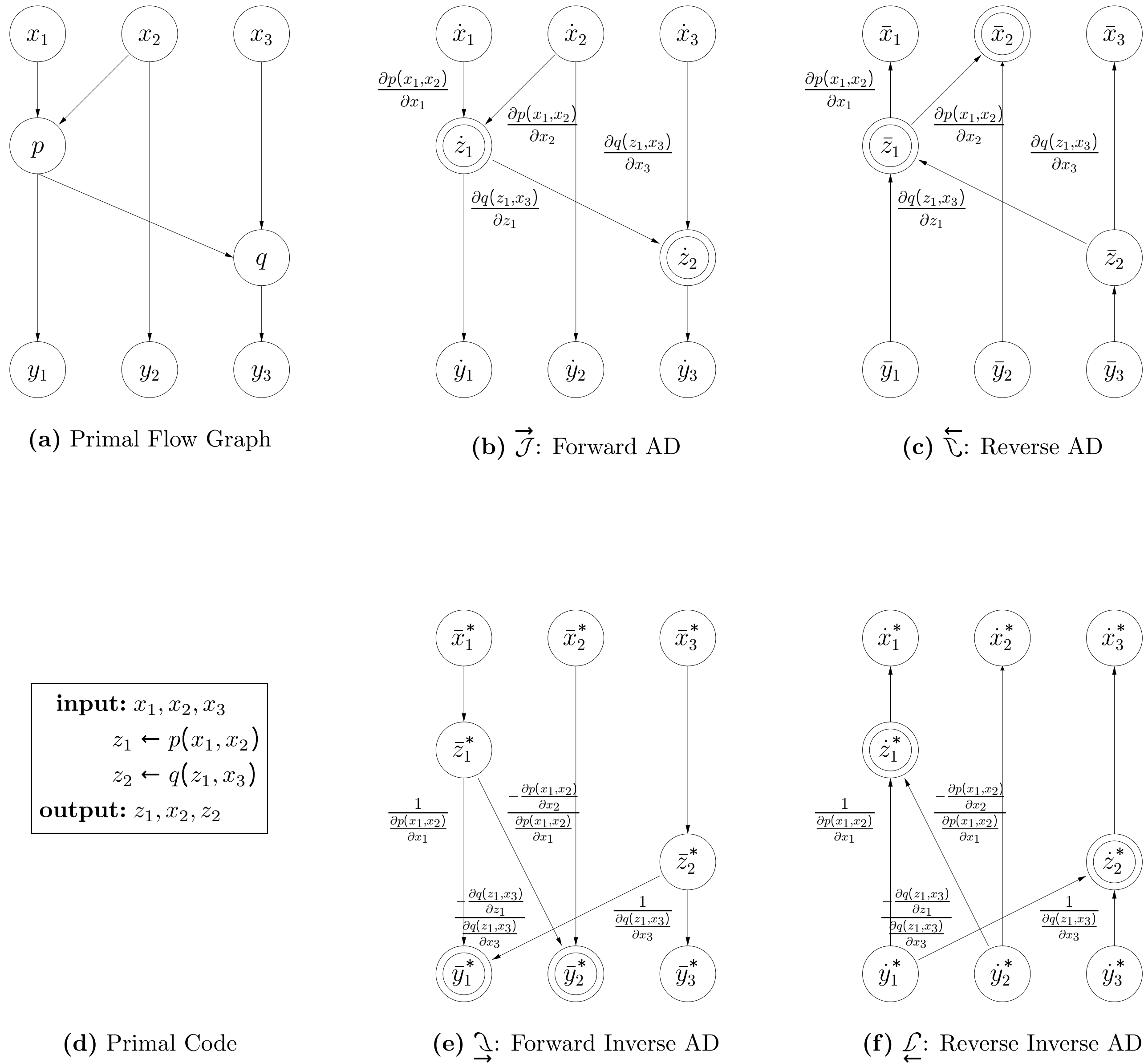
$\mathbf{J}^{-1}_{\mathbf{f}_t(\mathbf{x}_{t-1})} = \begin{pmatrix} \frac{1}{a} & \frac{-b}{a} \\ 0 & 1 \end{pmatrix}$; $\dot{\mathbf{x}}^* = \mathbf{J}^{-1}\dot{\mathbf{y}}^*$

- forward graph structure reversed
- different edge labels

Figure 2.1: Graphical representation of transformation of computation graph of binary atomic program step, for all four AD modes discussed. These are formulated for scalar inputs and outputs. In the case where the first input/output is a vector of length l and the second input is a vector of length $k-l$, one simply replaces a with $\mathbf{A}$, b with $\mathbf{B}$, the fraction $\frac{1}{a}$ with $\mathbf{A}^{-1}$, the fraction $\frac{-b}{a}$ with $-\mathbf{A}^{-1}\mathbf{B}$, the symbol 0 with $\mathbf{0}$, and 1 with $\mathbf{I}$.

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

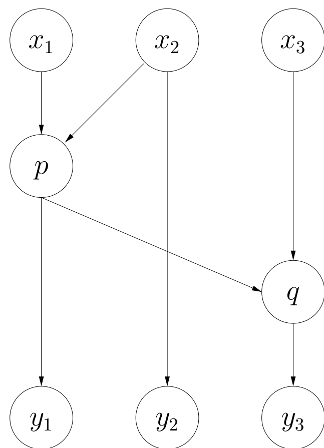


(a) Primal Flow Graph **(b)** $\overrightarrow{\mathcal{J}}$: Forward AD **(c)** $\overleftarrow{\mathcal{J}}$: Reverse AD

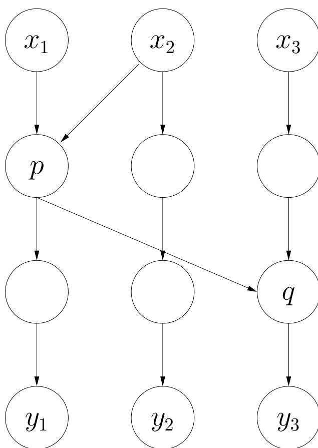
(d) Primal Code **(e)**: Forward Inverse AD **(f)**: Reverse Inverse AD

Figure 2.2: Illustration of all four AD modes for the straight-line code in (d). This corresponds to the data flow graph (a). The intent is that there are three registers, r_1, r_2, and r_3, illustrated by the three columns in (a) from left to right. These are initialized with x_1, x_2, and x_3 respectively. Since r_1 is not used after the first line of code, it is overwritten with z_1. Since r_3 is not used after the second line of code, it is overwritten with z_2. Forward mode and reverse mode are shown in (b) and (c) respectively. In these graphs, addition occurs whenever there is fan in to a vertex (the circled vertices) and labels on edges denote multiplication by the indicated coefficient. Reverse mode is derived from forward mode by edge reversal, which can change which vertices perform addition due to fan in. Forward inverse mode and reverse inverse mode are shown in (e) and (f) respectively. These have the same vertices as forward mode and reverse mode but different edges and edge labels, which changes which vertices perform addition due to fan in. Again, forward inverse mode is derived from reverse inverse mode by edge reversal.

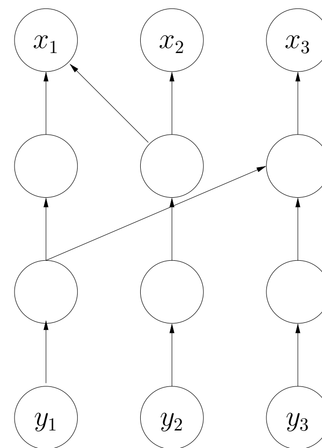
Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited



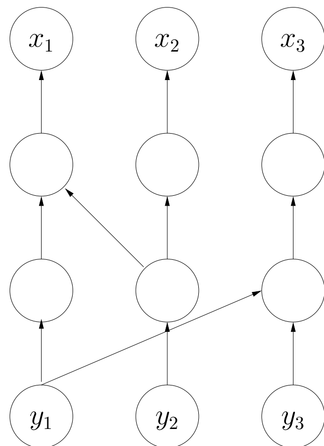
(a) Primal Flow Graph



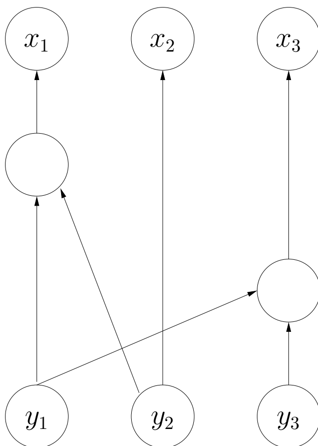
(b) Layering



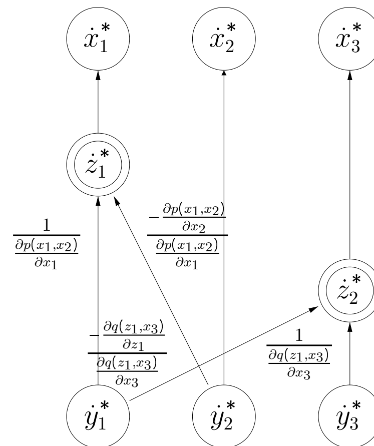
(c) Reverse-Inverse Transformation



(d) Time Shifting



(e) Unlayering



(f) Reverse Inverse Flow Graph

Figure 2.3: Illustration of the derivation of Figure 2.2(f) from Figure 2.2(a). Panel (a) corresponds to Figure 2.2(a). Panel (b) corresponds to construction of a layered flow graph [15] by carrying live variables forward. Panel (c) corresponds to applying the transformation of Figure 2.1(e). Panel (d) corresponds to shifting each operation one time step earlier to eliminate the noop in the first time step. Panel (e) corresponds to removing the layering. Panel (f) corresponds to Figure 2.2(f). Figure 2.2(b,c) are derived from Figure 2.2(a) using standard AD methods. Figure 2.2(c) is derived from Figure 2.2(b) using edge reversal. Figure 2.2(e) is derived from Figure 2.2(f) using edge reversal.

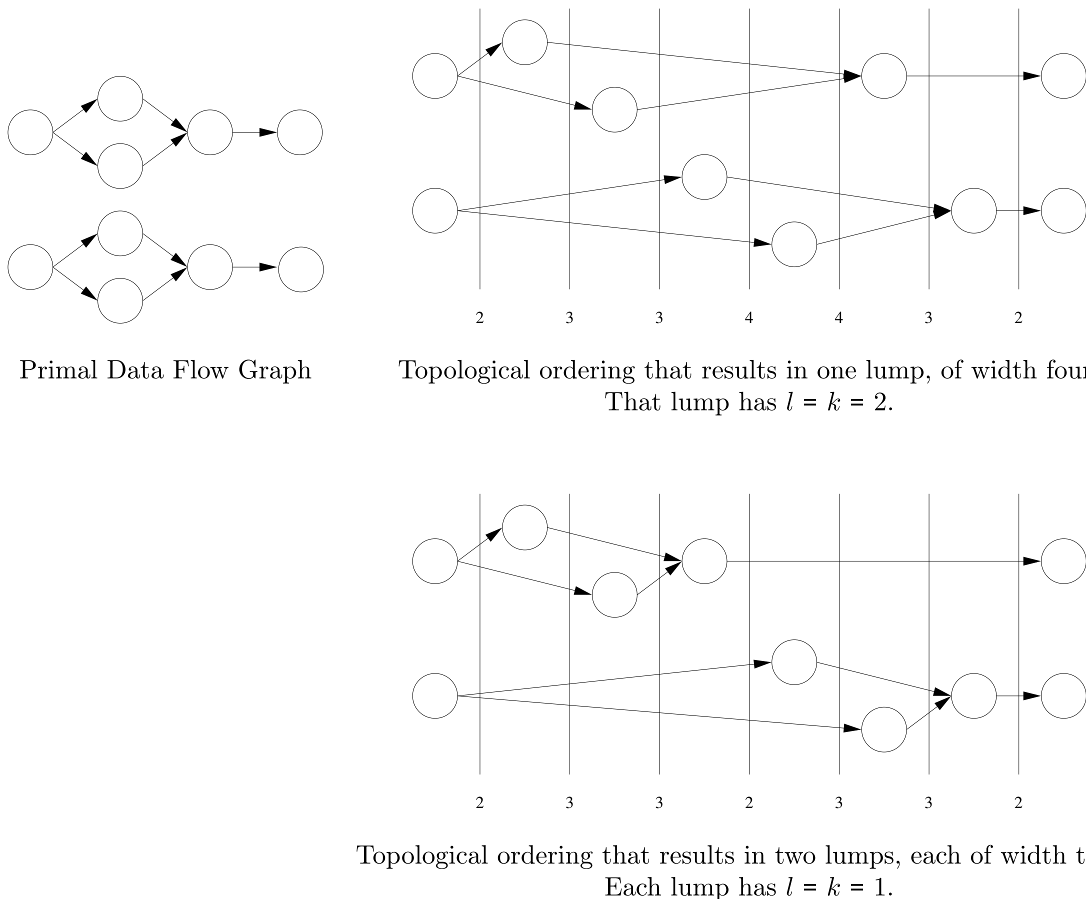
Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

3 Constant width graph. The limitation of the methods proposed above is that they require the computation to be constant width. What that means is that when the overall function is $\mathbb{R}^n \to \mathbb{R}^n$, there are precisely n live active variables (active in the AD sense) at each intermediate point between computation steps. This requires that every output value of a computation step overwrite some input value. Not all programs have this property. Some programs have varying numbers of live active variables as the computation proceeds, *i.e.*, temporary variables. If ever the number of live active variables is less than n, the Jacobian of the computation is necessarily singular.

But if ever the number of live active variables is greater than n, the computation can be partitioned into "lumps" where the number of live active variables between lumps is n. Each lump can be treated as a macro step and processed according to (2.4). The question then reduces to how to best partition a computation into lumps, and whether when doing so l is small. This process can be viewed as performing a topological sort of the computation graph, and then breaking that fully-ordered computation up into "lumps" at points where there are exactly n live variables. But there are many possible topological sorts, which may break the computation into different numbers of lumps of different sizes, as shown in Figure 3.1. We conjecture that a "greedy" algorithm will perform optimally here, but proving this conjecture is left for future work. We have implemented a system using preliminary answers to this question, which will be exhibited when its efficiency properties have been further explored.



Primal Data Flow Graph

Topological ordering that results in one lump, of width four. That lump has $l = k = 2$.

Topological ordering that results in two lumps, each of width three. Each lump has $l = k = 1$.

Figure 3.1: Illustration of lumpification's dependence on how a total ordering is imposed on the partially-ordered data-flow graph.

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

4 Notation. We have a bit of a combinatorial explosion of AD modes on our hands: forward *vs.* reverse, and noninverse *vs.* inverse, yielding four modes. To reason about these more easily, and facilitate their inclusion as first-class operators in programming languages, we propose notation which is short and mnemonic. Using a horizontal flip to indicate forward *vs.* reverse and a vertical one to indicate inverse, with a calligraphic J for Jacobian surmounted by an arrow to indicate flow, we have:

	forward	reverse
noninverted	$\overrightarrow{\mathcal{J}}\ f\ x\ \dot{x} \triangleq \mathbf{J}_{f(x)}\,\dot{x}$	$\overleftarrow{\reflectbox{$\mathcal{J}$}}\ f\ x\ \bar{y} \triangleq \mathbf{J}_{f(x)}^{\mathsf{T}}\,\bar{y}$
inverted	$\underrightarrow{\scalebox{1}[-1]{$\mathcal{J}$}}\ f\ x\ \bar{x}^{*} \triangleq \mathbf{J}_{f(x)}^{-\mathsf{T}}\,\bar{x}^{*}$	$\underleftarrow{\rotatebox{180}{$\mathcal{J}$}}\ f\ x\ \dot{y}^{*} \triangleq \mathbf{J}_{f(x)}^{-1}\,\dot{y}^{*}$

which obey a number of algebraic invariants:

$$(\overleftarrow{\reflectbox{$\mathcal{J}$}}\ f\ x\ \bar{y}) \cdot \dot{x} = \bar{y} \cdot (\overrightarrow{\mathcal{J}}\ f\ x\ \dot{x}) \tag{4.1a}$$

$$\bar{x}^{*} \cdot (\underleftarrow{\rotatebox{180}{$\mathcal{J}$}}\ f\ x\ \dot{y}^{*}) = (\underrightarrow{\scalebox{1}[-1]{$\mathcal{J}$}}\ f\ x\ \bar{x}^{*}) \cdot \dot{y}^{*} \tag{4.1b}$$

$$\overrightarrow{\mathcal{J}}\ f\ x \circ \underleftarrow{\rotatebox{180}{$\mathcal{J}$}}\ f\ x = \underrightarrow{\scalebox{1}[-1]{$\mathcal{J}$}}\ f\ x \circ \overleftarrow{\reflectbox{$\mathcal{J}$}}\ f\ x = \mathbf{id} \tag{4.1c}$$

$$\overleftarrow{\reflectbox{$\mathcal{J}$}}\ f\ x \circ \underrightarrow{\scalebox{1}[-1]{$\mathcal{J}$}}\ f\ x = \underleftarrow{\rotatebox{180}{$\mathcal{J}$}}\ f\ x \circ \overrightarrow{\mathcal{J}}\ f\ x = \mathbf{id}. \tag{4.1d}$$

5 Inverse automatic differentiation of an ordinary differential equation. Consider a primal computation $x_0 \mapsto x_T$ where $x(T_0) = x_0$ is the initial condition of an ordinary differential equation (ODE)

$$\frac{\mathrm{d}}{\mathrm{d}t} x = g(x) \tag{5.1}$$

and $x_T = x(T_1)$ is its final condition, the result of integrating (5.1) from T_0 to T_1. (If we wish to give g side parameters we can concatenate them onto x and extend g to give them zero derivatives, thus incorporating them into the current treatment without loss of generality. Alternatively, these can be treated as constants rather than active variables, so as non-active variables they do not enter into the constant-width calculation.)

We will discretize (5.1) with time-step $\Delta t > 0$, AD-transform the discretized system (using all four modes under consideration, using an approximation for the inverse of a near-identity matrix), and take the limit as $\Delta t \to 0$, yielding ODEs for the four modes. We use

$$x_k = x(T_0 + k\Delta t),$$

so the Euler approximation

$$x(t + \Delta t) = x(t) + \Delta t \frac{\mathrm{d}}{\mathrm{d}t} x(t)$$

becomes

$$x_k = f_k(x_{k-1}),$$

where

$$f_k(x) = x + \Delta t \cdot g(x).$$

This situation makes for step-wise Jacobians

$$\mathbf{J}_k = \mathbf{J}_{f_k(x_k)} = \mathbf{I} + \Delta t \cdot \mathbf{J}_{g(x_k)}$$

which determine the steps of the four AD accumulation modes under consideration,

$$\dot{x}_{k+1} = \mathbf{J}_k\,\dot{x}_k = \dot{x}_k + \Delta t \cdot (\overrightarrow{\mathcal{J}}\ g\ x_k\ \dot{x}_k) \tag{5.2a}$$

$$\bar{x}_k = \mathbf{J}_k^{\mathsf{T}} \bar{x}_{k+1} = \bar{x}_{k+1} + \Delta t \cdot (\overleftarrow{\reflectbox{$\mathcal{J}$}}\ g\ x_k\ \bar{x}_{k+1}) \tag{5.2b}$$

$$\dot{x}_k^{*} = \mathbf{J}_k^{-1} \dot{x}_{k+1}^{*} = \dot{x}_{k+1}^{*} - \Delta t \cdot (\overrightarrow{\mathcal{J}}\ g\ x_k\ \dot{x}_{k+1}^{*}) \tag{5.2c}$$

$$\bar{x}_{k+1}^{*} = \mathbf{J}_k^{-\mathsf{T}} \bar{x}_k^{*} = \bar{x}_k^{*} - \Delta t \cdot (\overleftarrow{\reflectbox{$\mathcal{J}$}}\ g\ x_k\ \bar{x}_k^{*}). \tag{5.2d}$$

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Using the identity $(\mathbf{I} + \Delta t\mathbf{A})^{-1} = \mathbf{I} - \Delta t\mathbf{A} + O(\Delta t^2)$ gives per-step errors of $O(\Delta t^2)$, which over the course of $O(1/\Delta t)$ time-steps gives a final numeric error of $O(\Delta t)$. Putting these in the form $(v_{k+1} - v_k)/\Delta t = b$ and taking the limit $\Delta t \to 0$,

$$\frac{\mathrm{d}}{\mathrm{d}t}\dot{x} = \overrightarrow{\mathcal{J}}\ g\ x\ \dot{x} \qquad \dot{x}(T_0) \mapsto \dot{x}(T_1) \tag{5.3a}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\bar{x} = -\overleftarrow{\rotatebox[origin=c]{90}{$\mathcal{J}$}}\ g\ x\ \bar{x} \qquad \bar{x}(T_0) \mapsfrom \bar{x}(T_1) \tag{5.3b}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\dot{x}^* = \overrightarrow{\mathcal{J}}\ g\ x\ \dot{x}^* \qquad \dot{x}^*(T_0) \mapsfrom \dot{x}^*(T_1) \tag{5.3c}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\bar{x}^* = -\overleftarrow{\rotatebox[origin=c]{90}{$\mathcal{J}$}}\ g\ x\ \bar{x}^* \qquad \bar{x}^*(T_0) \mapsto \bar{x}^*(T_1). \tag{5.3d}$$

Note that (5.3a) and (5.3c) are identical, as are (5.3b) and (5.3d), except that the specified/calculated boundary condition differs in location between T_0 and T_1, so the direction of integration is reversed. If the primal equation (5.1) is stable, the linear ODEs (5.3a) and (5.3b) are also stable, and therefore (5.3c) and (5.3d) would be unstable. This is an intrinsic property: if a linear operator is stable its inverse will be unstable, since the eigenvalues are inverted.

In a system which allows AD transforms of basis functions to be user-specified, and which has higher-order functions including differential equation solvers, this would suggest efficient direct transforms of such solvers not just for forward and reverse AD, as is now routine (see for example the Diffrax[1] subsystem [8] for the AD-enabled language JAX, or torchode[2] for PyTorch [10]) but also for inverse AD.

6 Implementation. We have developed a prototype implementation for inverse AD of constant-width computations. Work is currently underway to automatically detect "lumps" and handle them appropriately. This problem is more difficult than it might initially appear, because the computation graph can be broken up in different ways. To avoid treating this as a brute-force combinatorial problem, either heuristics must be employed, or connections to efficient graph algorithms like max flow must be made. See section 3 for further discussion of this issue.

7 Related work.

7.1 Classic work on the inverse problem. The idea of direct calculation of the solution of a linear system resulting from the linearization of a function represented as a computer program was introduced by [6] and elaborated by [4], [21], and [7, Chapter 4], using a framework in which the multiple $\mathbf{J}_t$ matrices here are replaced by a single much larger matrix. That framework is quite general, but requires that the computation graph be stored and manipulated in a fashion which seems difficult to migrate to compile-time. At root this is because that formulation is not compositional. The present framework, which is compositional, is amenable to efficient implementation, which we have done in a preliminary implementation.

7.2 Recent related work. An early version of this work was publicly discussed in 2019[3] and its implementation in JAX was proposed in 2022 by Neil Girdhar and discussed at length,[4] and added and removed from JAX proper by Matt Johnson.[5] Reference [15] discussed some of the ideas presented in that work and here, but did not cite the earlier work just discussed. It also does not contain equations (2.2), (2.3), or (2.4), does not discuss the fact that simple operations that preserve width result in $\mathbf{A} : l \times l$ and $\mathbf{B} : l \times (k - l)$, does not discuss the fact that although $\mathbf{J}_t$ is not structurally symmetric, $\mathbf{J}_t^{-1}$ has the same structural sparsity as $\mathbf{J}_t$ and therefore the amount of arithmetic is the same as for conventional Forward and Reverse modes, but that these are transposed so Forward Inverse Mode writes to the derivative-related quantities associated with *all* involved variables of each basis function invocation, while Reverse Inverse Mode writes only to the quantities associated with slots *written to* in the primal computation of each basis function, does not discuss "lumpification," does not

[1] https://docs.kidger.site/diffrax/
[2] https://github.com/martenlienen/torchode
[3] https://openreview.net/forum?id=Bygj2Ys6IS https://openreview.net/pdf?id=Bygj2Ys6IS
[4] https://github.com/jax-ml/jax/issues/12494
[5] https://github.com/jax-ml/jax/commit/902fc0c3d2b3ec9b6034c66074984386ec35606f

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited
Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

show a compositional method analogous to Figures 2.1 to 2.3, does not discuss the invariants (4.1), and does not discuss issues of stability of inverse AD, *e.g.*, of inverse AD of a stable ODE.

Naumann [15] does explore a number of technical issues not discussed here. It introduces a formulation in which "pass through" nodes are added to the computation graph, so that width is defined in terms of node cuts rather than edge cuts. It also shows that a variety of scheduling issues associated with inverse AD are NP-complete.

The NP-hardness result might not impact the formulation here as it seeks to determine the minimal number of arithmetic operations to perform inverse AD. In the case with basis functions with multiple outputs, as would arise when coalescing lumps, this would involve selecting a lumpification that made $\mathbf{A}$ and $\mathbf{B}$ suitably sparse to minimize the number of arithmetic operations to compute $\mathbf{A}^{-1}$ and $-\mathbf{A}^{-1}\mathbf{B}$. If one was not concerned with the sparsity of these operations, treating them as atomic, and one was interested only in doing forward inverse AD or reverse inverse AD to compute inverse Jacobian (transpose) vector products, and not some hybrid that involved vertex, edge, or face elimination to compute the full inverse Jacobian, the requisite lumpification process reduces to selecting a topological sort that minimizes maximal lump size, maximal lump width, or maximal values of l and k. We conjecture that a greedy algorithm may suffice for this.

7.3 Linguistic support and program inversion. Invertible computation in general, and automatic program inversion in particular, has been a subject of study in programming language theory for decades [5], with applications in protocol design, automatic model updates, *etc*. This is closely related to a special case of the present formulation, namely fully invertible numeric computations. The idea of program inversion has been pursued by a small but dedicated community, with some striking results [13, 14, 9].

Here we have focused on a computation which is locally invertible, meaning that its linearization around a point is invertible. This property necessarily holds when a computation is globally invertible and also differentiable. Techniques developed there for statically guaranteeing complete invertibility using a type system may be applicable here as well, to guarantee local invertibility: guaranteeing by construction that the data flow graph of active variables will be constant width.

We also note that it is well known that if the primal is stepwise invertible then the tape normally used in reverse mode to store values computed during the forward sweep for use in the reverse sweep in reverse order can be eliminated as these values can be recomputed in reverse order during the reverse sweep [18, 11]. Analogously, under this constraint the tape can also be eliminated from reverse-inverse mode.

7.4 Quantum computing and machine learning. Quantum computation is also invertible and differentiable in the precise sense used in this manuscript. On a topical note, invertible differentiable programs form the kernels of a variety of generative AI systems: methods like BS-Infomax [2] and its context-sensitive generalizations like cICA [17, 19], monotonic neural networks [22], normalizing flows [16], and stable diffusion [20], have at their hearts multi-layer structures carefully constructed to be invertible, along with manually-derived inversion procedures. Bread-and-butter deep learning models like ResNet [1] or CNNs frequently used in deep learning can be easily modified to maintain a constant width, particularly if re-cast as differential equations [12]. Efficient support for locally invertible numeric computations, using inverse AD, would allow models of the above classes to be generalized and much more easily implemented, as the extremely tedious and error-prone manual derivations and coding would be avoided.

Acknowledgments. This research was supported, in part, by US National Science Foundation (NSF) grants 1522954-IIS and 1734938-IIS, by a US Intelligence Advanced Research Projects Activity (IARPA) grant via Department of Interior/Interior Business Center (DOI/IBC) contract number D17PC00341, by Taighde Éireann — Research Ireland under grant number 20/FFP-P/8853, and by the Defense Advance Research Projects Agency (prime contract award HR0011222003, subcontract award 2103299-01, grant 13001129). For the purpose of Open Access, the authors have applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission. The content of the information does not necessarily reflect the position of the US Government. No official endorsement should be inferred. Approved for public release; distribution is unlimited.

References

[1] J. Behrmann, W. Grathwohl, R. T. Q. Chen, D. Duvenaud, and J.-H. Jacobsen, *Invertible residual networks*, in Proceedings of the 36th International Conference on Machine Learning, K. Chaudhuri

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

and R. Salakhutdinov, eds., vol. 97 of Proceedings of Machine Learning Research, Long Beach, California, USA, 2019, PMLR, pp. 573–582, https://doi.org/10.48550/arXiv.1811.00995.

[2] A. J. Bell and T. J. Sejnowski, *An information-maximization approach to blind separation and blind deconvolution*, Neural Computation, 7 (1995), pp. 1129–1159, https://doi.org/10.1162/neco.1995.7.6.1129.

[3] B. Bullins, K. Patel, O. Shamir, N. Srebro, and B. E. Woodworth, *A stochastic Newton algorithm for distributed convex optimization*, in Advances in Neural Information Processing Systems 34, Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS 2021), M. Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Liang, and J. W. Vaughan, eds., Red Hook, NY, 2021, Curran Associates, pp. 26818–26830, https://doi.org/10.48550/arXiv.2110.02954.

[4] L. C. W. Dixon, *Use of automatic differentiation for calculating Hessians and Newton steps*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, PA, 1991, pp. 114–125.

[5] D. Gries, *Inverting programs*, in The Science of Programming, Springer, New York, NY, 1981, ch. 21, pp. 265–274, https://doi.org/10.1007/978-1-4612-5983-1_22.

[6] A. Griewank, *Direct calculation of Newton steps without accumulating Jacobians*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., SIAM, Philadelphia, Penn., 1990, pp. 115–137.

[7] A. S. Hossain, *On the Computation of Sparse Jacobian Matrices and Newton Steps*, PhD thesis, Department of Informatics, University of Bergen, 1998. Technical Report 146.

[8] P. Kidger, *On Neural Differential Equations*, PhD thesis, Mathematical Institute, University of Oxford, 2021, https://doi.org/10.48550/arXiv.2202.02435.

[9] J. T. Kristensen, R. Kaarsgaard, and M. K. Thomsen, *Jeopardy: An invertible functional programming language*, in Reversible Computation, T. Æ. Mogensen and Ł. Mikulski, eds., Cham, 2024, Springer Nature Switzerland, pp. 124–141, https://doi.org/10.1007/978-3-031-62076-8_9.

[10] M. Lienen and S. Günnemann, *torchode: A parallel ODE solver for PyTorch*, in The Symbiosis of Deep Learning and Differential Equations II, Workshop at NeurIPS 2022, 2022, https://doi.org/10.48550/arXiv.2210.12375.

[11] D. Maclaurin, D. Duvenaud, and R. Adams, *Gradient-based hyperparameter optimization through reversible learning*, in Proceedings of the 32nd International Conference on Machine Learning, F. Bach and D. Blei, eds., vol. 37 of Proceedings of Machine Learning Research, Lille, France, 2015, PMLR, pp. 2113–2122, https://doi.org/10.48550/arXiv.1502.03492.

[12] M. Maleki, M. Habiba, and B. A. Pearlmutter, *HeunNet: Extending ResNet using Heun's method*, in 32nd Irish Signals and Systems Conference (ISSC), 2021, pp. 1–6, https://doi.org/10.1109/ISSC52156.2021.9467884.

[13] K. Matsuda, S.-C. Mu, Z. Hu, and M. Takeichi, *A grammar-based approach to invertible programs*, in Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20–28, 2010. Proceedings, A. D. Gordon, ed., vol. 6012 of Lecture Notes in Computer Science, Springer, 2010, pp. 448–467, https://doi.org/10.1007/978-3-642-11957-6_24.

[14] K. Matsuda and M. Wang, *Sparcl: A language for partially-invertible computation*, Proceedings of the ACM on Programming Languages, 4 (2020), pp. 118:1–118:31, https://doi.org/10.1145/3409000.

[15] U. Naumann, *A matrix-free exact Newton method*, SIAM Journal on Scientific Computing, 46 (2024), pp. A1423–A1440, https://doi.org/10.1137/23M157017X.

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

[16] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan, *Normalizing flows for probabilistic modeling and inference*, Journal of Machine Learning Research, 22 (2021), pp. 1–64, https://jmlr.org/papers/volume22/19-1028/19-1028.pdf.

[17] L. Parra, G. Deco, and S. Miesbach, *Redundancy reduction with information-preserving nonlinear maps*, Network: Computation in Neural Systems, 6 (1995), pp. 61–72, https://doi.org/10.1088/0954-898X_6_1_004.

[18] B. A. Pearlmutter, *Gradient calculations for dynamic recurrent neural networks: A survey*, IEEE Transactions on Neural Networks, 6 (1995), pp. 1212–1228, https://doi.org/10.1109/72.410363.

[19] B. A. Pearlmutter and L. C. Parra, *A context-sensitive generalization of ICA*, in International Conference on Neural Information Processing, Hong Kong, Sept. 24–27, 1996, Springer, 1996, pp. 151–157.

[20] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, *High-resolution image synthesis with latent diffusion models*, in 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2022, pp. 10674–10685, https://doi.org/10.1109/CVPR52688.2022.01042.

[21] J. Utke, *Efficient Newton steps without Jacobians*, in Computational Differentiation: Techniques, Applications, and Tools, M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, eds., SIAM, Philadelphia, PA, 1996, pp. 253–264.

[22] A. Wehenkel and G. Louppe, *Unconstrained monotonic neural networks*, in Advances in Neural Information Processing Systems 32, Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, eds., Red Hook, NY, 2019, Curran Associates, pp. 1534–1544, https://doi.org/10.48550/arXiv.1908.05164.

Downloaded 03/01/26 to 174.219.213.35 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited