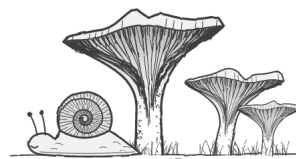


When the Jacobian Is Dense but the Program Is Not

Mikkel Paltor (mpasc@dtu.dk), 23 October 2026

Department of Applied Mathematics and Computer Science, Technical University of Denmark



In machine learning, automatic differentiation is mostly used for products with the Jacobian: the gradient of a loss is a single vector–Jacobian product, and training never forms the Jacobian itself. Many methods in scientific computing need the matrix. Newton’s method for a nonlinear system $F(x) = 0$, at the heart of implicit time stepping for stiff differential equations and of steady-state solvers for discretized partial differential equations, factorizes the Jacobian in every step, and solvers for constrained optimization, such as Ipopt [1] and Uno [2], take the Jacobian of the constraints and the Hessian of the Lagrangian as matrices. These Jacobians are large, with thousands to millions of unknowns, but more often than not sparse, since each equation involves only a few unknowns.

An Illustrated Guide to Automatic Sparse Differentiation¹ [3] explains how standard automatic differentiation tools can be used to compute a sparse Jacobian at a fraction of the cost of a dense one. The idea is fairly simple: first detect which entries can be nonzero, group the columns (or rows) that never share a row (or column), and compute Jacobian–vector (or vector–Jacobian) products in order to generate the sparse Jacobian.² In Backpropagation as a Linear Solve³ we derived that the Jacobian of *any* computation can be computed as

$$J = PL^{-1}M^{\top}, \quad (1)$$

where P , M , L are matrices which more often than not are *sparse* and with L further being unit lower triangular. Even when J is dense, products with it are cheap: $Jv = P(L^{-1}(M^{\top}v))$ is evaluated from right to left, with one forward substitution in the sparse, unit lower triangular L , at about the cost of running the program.⁴ What a dense J makes expensive is forming it, or solving with it.

This post asks what the matrices L , M , and P add to the understanding of the Jacobian. The answer in short is that they pay off when L is sparse but L^{-1} is dense, resulting in a dense Jacobian J (e.g., Example 1).

Example 1 (A cumulative sum) *We will explain how to compute P , L , and M in Recap 1, but for now just trust the process.*

For the scaled cumulative sum of exponentials, $f_k(x) = p_k \sum_{j \leq k} \exp(-x_j)$ with constants p_k , computed by $u_k = u_{k-1} + \exp(-x_k)$, L is lower bidiagonal, and $M^{\top} = -\text{diag}(m)$ with $m_j = \exp(-x_j)$ and $P = \text{diag}(p)$ are diagonal. The inverse of L is dense,

Contents

1. From program to matrices
 2. The augmented system
 3. Lifting
 4. Summary
- Bibliography

¹ ICLR 2025 blog post by Hill, Dalle, and Montoison.

iclr-blogposts.github.io/2025/blog/sparse-autodiff/

² The grouping is for efficiency: all columns of a group are computed by a single Jacobian–vector product, and all rows of a group by a single vector–Jacobian product.

³ Earlier post: *Backpropagation as a Linear Solve*. mipals.github.io/blog/2026/10-05-backpropagation-linear-solve/

⁴ Likewise for the transpose: $J^{\top}w = M(L^{-\top}(P^{\top}w))$ needs one backward substitution in the unit upper triangular $L^{\top}$, which is reverse mode.

Example 1: A cumulative sum

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \quad (2)$$

and so is the Jacobian $J = PL^{-1}M^\top$. However, often the dense Jacobian need not be explicitly formed. A product with J , or a solve with it, can be written as one sparse system,

$$y = Jv \iff \begin{bmatrix} L & -M^\top \\ P & 0 \end{bmatrix} \begin{bmatrix} z \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix}, \quad (3)$$

whose first block row gives $z = L^{-1}M^\top v$ and second $y = Pz = Jv$. This *augmented system* keeps the sparsity of L , M , and P however dense J is, and it is the subject of this post.

Recap 1: The three matrices

Recap 1 (The three matrices) This recaps the derivation of Backpropagation as a Linear Solve⁵, written for a general program instead of a network. A program computes $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ by a sequence of elementary operations, such as a product, a sum, or an exponential. Each operation in any program produces an *intermediate variable* $u_i = \Phi_i(u_1, \dots, u_{i-1}, x)$ from earlier intermediate variables and the inputs x ⁶. Each intermediate variable becomes a *node* in the computational graph of the program, with an edge into node i from every u_j and x_k that Φ_i actually depends on. Collecting the d nodes gives the forward relation $u = \Phi(u, x)$. Most of the time we're only interested in a selection of the intermediate computation, so we introduce a matrix $P \in \mathbb{R}^{m \times d}$, usually a selection with entries 0 and 1, so that $f(x) = Pu$. Differentiating the forward relation gives $du = \frac{\partial \Phi}{\partial u} du + \frac{\partial \Phi}{\partial x} dx$, that is, $L du = M^\top dx$ with

$$L = I - \frac{\partial \Phi}{\partial u} \in \mathbb{R}^{d \times d}, \quad M^\top = \frac{\partial \Phi}{\partial x} \in \mathbb{R}^{d \times n}, \quad (4)$$

and therefore $J = PL^{-1}M^\top$. Since a node reads only earlier nodes, $\partial \Phi / \partial u$ is strictly lower triangular and L is unit lower triangular, and since it reads only a few of them, L and M have only a few nonzeros per row.

⁵ Earlier post: *Backpropagation as a Linear Solve*.
mipals.github.io/blog/2026/10-05-backpropagation-linear-solve/

⁶ That it only depends on previous nodes gives a directed acyclic graph, which we here represent as a lower triangular matrix.

Example 2: Four nodes

Example 2 (Four nodes) To compute $f(x_1, x_2) = (x_1 x_2 + \sin(x_1), \exp(x_2))$, a program introduces one intermediate variable per elementary operation, four in total: $u_1 = x_1 x_2$, $u_2 = \sin(x_1)$, $u_3 = u_1 + u_2$, and $u_4 = \exp(x_2)$. The forward relation and the outputs are

$$\underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}}_u = \underbrace{\begin{bmatrix} x_1 x_2 \\ \sin(x_1) \\ u_1 + u_2 \\ \exp(x_2) \end{bmatrix}}_{\Phi(u, x)}, \quad f(x) = \underbrace{\begin{bmatrix} u_3 \\ u_4 \end{bmatrix}}_P = \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_P \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}}_u. \quad (5)$$

Row i of Φ is the operation Φ_i that computes the intermediate variable u_i . Implicitly, Φ defines a computational graph, as every intermediate variable u_i depends on only

a few earlier u_j and inputs x_k . The Jacobian of Φ with respect to all of its arguments (u, x) splits into the two blocks of partial derivatives

$$\frac{\partial \Phi}{\partial u} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad M^\top = \frac{\partial \Phi}{\partial x} = \begin{bmatrix} x_2 & x_1 \\ \cos(x_1) & 0 \\ 0 & 0 \\ 0 & \exp(x_2) \end{bmatrix}, \quad (6)$$

The intermediate variables depend on x through the forward relation itself, $u(x) = \Phi(u(x), x)$, so the chain rule gives

$$\frac{du}{dx} = \frac{\partial \Phi}{\partial u} \frac{du}{dx} + \frac{\partial \Phi}{\partial x} \iff \underbrace{\left(I - \frac{\partial \Phi}{\partial u} \right)}_L \frac{du}{dx} = \underbrace{\frac{\partial \Phi}{\partial x}}_{M^\top}. \quad (7)$$

In our example the unit lower triangular matrix L is⁷

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad L^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (8)$$

Solving with L one intermediate variable at a time, by forward substitution, gives

$$\frac{du}{dx} = L^{-1} M^\top = \begin{bmatrix} x_2 & x_1 \\ \cos(x_1) & 0 \\ x_2 + \cos(x_1) & x_1 \\ 0 & \exp(x_2) \end{bmatrix}. \quad (9)$$

The Jacobian of $f(x) = Pu(x)$ picks out the rows of the outputs,

$$J = P \frac{du}{dx} = PL^{-1} M^\top = \begin{bmatrix} x_2 + \cos(x_1) & x_1 \\ 0 & \exp(x_2) \end{bmatrix}. \quad (10)$$

⁷ Writing $L = I - \hat{L}$ with $\hat{L} = \partial \Phi / \partial u$ strictly lower triangular, $\hat{L}$ is nilpotent, $\hat{L}^d = 0$, so the Neumann series of L^{-1} terminates, $L^{-1} = I + \hat{L} + \dots + \hat{L}^{d-1}$. Here $\hat{L}^2 = 0$, since u_3 depends only on intermediate variables that depend on no others, so $L^{-1} = I + \hat{L}$.

1. From program to matrices

We can create L , M , and P automatically by *recording* the program. A *tape* stores the partial derivatives of every operation, evaluated as the program runs; the intermediate values are only needed during the run, to compute those partial derivatives and the outputs. The tape of this post is deliberately small. A Tracked number holds a value and the index of its node. Arithmetic on Tracked numbers is overloaded, as for dual numbers: $a * b$ computes the product, gives it the next node index, and stores its partial derivatives, b with respect to a and a with respect to b , as entries of L or $M^\top$, depending on whether a and b are nodes or inputs. The function `trace` runs f on Tracked inputs and returns $f(x)$ together with the sparse matrices L , $M^\top$, and P .

Code 1: A scalar tape in Julia

Code 1 (A scalar tape in Julia)

julia

```
1 # A scalar tape. Every operation on a Tracked number is one node  $\Phi_i$ 
2 # of the forward relation  $u = \Phi(u, x)$ , and it stores its partial
3 # derivatives, the entries  $L[i, j] = -\partial \Phi_i / \partial u_j$  and  $M^\top[i, k] = \partial \Phi_i / \partial x_k$ .
4 using LinearAlgebra, SparseArrays
5
6 # The tape: rows, columns and values of the entries of  $L - I$  and  $M^\top$ ,
```

```

7 # and the number of nodes
8 const L_entries = (Int[], Int[], Float64[])
9 const Mt_entries = (Int[], Int[], Float64[])
10 const nodes = Ref{0}()
11
12 # index > 0 is the node ui, index < 0 is the input x-i
13 struct Tracked
14     value::Float64
15     index::Int
16 end
17
18 # Append the node with the given value and (argument => partial) pairs
19 function record!(value, args...)
20     i = nodes[] += 1
21     for (a, ∂a) in args
22         rows, cols, vals = a.index > 0 ? L_entries : Mt_entries
23         push!(rows, i)
24         push!(cols, abs(a.index))
25         push!(vals, a.index > 0 ? -∂a : ∂a)
26     end
27     return Tracked(value, i)
28 end
29
30 Base.:+(a::Tracked, b::Tracked) = record!(a.value + b.value, a => 1.0, b
=> 1.0)
31 Base.-:(a::Tracked, b::Tracked) = record!(a.value - b.value, a => 1.0, b
=> -1.0)
32 Base.:*(a::Tracked, b::Tracked) = record!(a.value * b.value, a => b.value,
b => a.value)
33 Base.:/(a::Tracked, b::Tracked) =
34     record!(a.value / b.value, a => 1 / b.value, b => -a.value /
b.value^2)
35 Base.:+(a::Tracked, c::Real) = record!(a.value + c, a => 1.0)
36 Base.:+(c::Real, a::Tracked) = a + c
37 Base.-:(a::Tracked, c::Real) = a + (-c)
38 Base.-:(c::Real, a::Tracked) = record!(c - a.value, a => -1.0)
39 Base.-:(a::Tracked) = record!(-a.value, a => -1.0)
40 Base.:*(a::Tracked, c::Real) = record!(a.value * c, a => Float64(c))
41 Base.:*(c::Real, a::Tracked) = a * c
42 Base.:/(a::Tracked, c::Real) = a * inv(c)
43 Base.:/(c::Real, a::Tracked) = record!(c / a.value, a => -c / a.value^2)
44 Base.exp(a::Tracked) = (v = exp(a.value); record!(v, a => v))
45 Base.sin(a::Tracked) = record!(sin(a.value), a => cos(a.value))
46 Base.:^(a::Tracked, p::Integer) = record!(a.value^p, a => p * a.value^(p -
1))
47
48 # Run f on tracked inputs and return f(x) with the sparse matrices
49 # L = I - ∂Φ/∂u, MT = ∂Φ/∂x and the selection P of the outputs, so
50 # that the Jacobian of f at x is P L-1 MT.
51 function trace(f, x)
52     foreach(empty!, (L_entries..., Mt_entries...))
53     nodes[] = 0
54     y = f([Tracked(xk, -k) for (k, xk) in enumerate(x)])
55     d, n, m = nodes[], length(x), length(y)
56     L = I + sparse(L_entries..., d, d) # sums the entries of x * x
57     Mt = sparse(Mt_entries..., d, n)
58     P = sparse(1:m, [yi.index for yi in y], ones(m), m, d)
59     return [yi.value for yi in y], L, Mt, P
60 end

```

With the matrices at hand, the Jacobian itself is a triangular solve with a sparse right-hand side, $J = P(L^{-1}M^T)$. Column k of $L^{-1}M^T$ is nonzero only at the intermediate variables that depend on x_k , and a sparse triangular solve finds those by a search through the graph of L before it does any arithmetic. SparseSparse.jl does

this when both sides of $\backslash$ are sparse, whereas the standard library returns a dense matrix.

Code 2: A Jacobian by a sparse solve

Code 2 (A Jacobian by a sparse solve)

julia

```
1 using SparseSparse # Required for \ with a sparse right-hand side
2 include("tape.jl")
3
4 # u''(x1) by central differences, with u = 0 at both ends
5 function cdiff2(x, i, h)
6     left = i > 1 ? x[i-1] : 0.0
7     right = i < length(x) ? x[i+1] : 0.0
8     return (left - 2x[i] + right) / h^2
9 end
10
11 # Bratu, u'' + λ exp(u) = 0: each output reads three inputs
12 function bratu(x; λ=1.0)
13     h = 1 / (length(x) + 1)
14     return [cdiff2(x, i, h) + λ * exp(x[i]) for i in eachindex(x)]
15 end
16
17 # Nonlocal Ohmic heating, u'' + λ exp(-u) / (∫01 exp(-u) dx)2 = 0, of the
18 # next section: every output reads the sum s, so every input
19 function heating(x; λ=2.0)
20     h = 1 / (length(x) + 1)
21     s = sum(exp(-xj) for xj in x) # h * s is the integral
22     scaling = λ / (h * s)^2
23     return [cdiff2(x, i, h) + scaling * exp(-x[i]) for i in eachindex(x)]
24 end
25
26 x = 0.1 .* rand(1000)
27 _, L, Mt, P = trace(bratu, x) # nnz(L) = 13000, nnz(M) = 3998
28 J = P * (L \ Mt) # tridiagonal: nnz(J) = 2998
29 _, L, Mt, P = trace(heating, x) # nnz(L) = 27997, nnz(M) = 4998
30 J = P * (L \ Mt) # dense: nnz(J) = 1000000
```

2. The augmented system

While both the Bratu and the heating residual (Example 3) in Code 2 give sparse P , L , and M , only the Bratu Jacobian is sparse. The cumulative sum of Example 1 already showed how this can happen: a sparse L can have a dense inverse, in which case *sparse differentiation cannot help*. For products, a dense J is no problem, since $Jv = P(L^{-1}(M^\top v))$ costs one forward substitution. Newton's method for $F(x) = 0$, however, needs a solve, $J\Delta = F(x)$, in every step, and with a dense J that means forming J and factorizing it, at a cost of order n^3 . The cumulative sum also hinted at the way around this: given P , L , and M , the solve can be written as one sparse *augmented* system,

$$J\Delta = F(x) \iff \underbrace{\begin{bmatrix} L & -M^\top \\ P & 0 \end{bmatrix}}_A \begin{bmatrix} z \\ \Delta \end{bmatrix} = \begin{bmatrix} 0 \\ F(x) \end{bmatrix}, \quad (11)$$

whose first block row gives $z = L^{-1}M^\top\Delta$ and second $Pz = J\Delta = F(x)$. Newton's method can therefore solve with the sparse A , whose unknowns are z and the step Δ , instead of with the dense J .⁸

⁸ The same factorization of A also solves with $J^\top$, as in adjoint methods: $J^\top w = g$ is $\begin{bmatrix} L^\top & P^\top \\ -M & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ g \end{bmatrix}$, the system with $A^\top$.

Example 3 (Ohmic heating) A nonlocal model of a resistor heated by a current [4] is

$$u'' + \frac{\lambda \exp(-u)}{\left(\int_0^1 \exp(-u) dx\right)^2} = 0, \quad u(0) = u(1) = 0. \quad (12)$$

With the integral approximated by hs with $s = \sum_j \exp(-x_j)$, the residual is

$$F_i(x) = \frac{x_{i-1} - 2x_i + x_{i+1}}{h^2} + \frac{\lambda \exp(-x_i)}{(hs)^2}. \quad (13)$$

Every F_i depends on every x_j through s , so J is dense. However, it has plenty of structure as it is a tridiagonal matrix plus the rank-one term $\frac{2\lambda}{h^2 s^3} ee^\top$ with $e_i = \exp(-x_i)$. Any reasonable hand-written solver would use the Sherman–Morrison formula to solve this.⁹ The augmented system finds the same structure without being told, since the tape contains the chain of sums that computes s . A Newton step is then¹⁰

```
A = [L -Mt; P spzeros(m, n)]
Δ = (A \ [zeros(d); F])[d+1:end]
```

n	$\text{nnz}(A)$	Step
500	16 995	4.2 ms
1000	33 995	8.1 ms
2000	67 995	19 ms
4000	135 995	46 ms
8000	271 995	0.14 s

Example 4 (The heating problem in JAX) JAX, like other standard automatic differentiation software, not record L and M . `jax.linearize(f, x)` evaluates f and its partial derivatives once and returns a *linearizer*, the map $v \mapsto P(L^{-1}(M^\top v))$ as a program (a *jaxpr*). The program never forms L^{-1} rather it uses the unit lower triangular structure of $L = I - \hat{L}$. Since $\hat{L}$ is strictly lower triangular, each row needs only the rows before it,

$$t_i = (M^\top v)_i + \sum_{j < i} \hat{L}_{ij} t_j, \quad i = 1, \dots, d, \quad (14)$$

so computing the rows in order gives t in one sweep. Each step applies row i of $M^\top$ and of $\hat{L}$ at once, so L and M are *fused* into a single program. It can be transposed for reverse mode, $w \mapsto M(L^{-\top}(P^\top w))$, but not taken apart into the matrices that the augmented system needs. A Newton step for the heating problem therefore has to form the dense J , with n Jacobian–vector products (`jax.jacfwd`) or n vector–Jacobian products (`jax.jacrev`), and factorize it densely.¹¹

Example 3: Ohmic heating

⁹ With $J = T + \gamma ee^\top$, T tridiagonal, and $\gamma = 2\lambda/(h^2 s^3)$, we have that

$$J^{-1}b = T^{-1}b - \frac{\gamma e^\top T^{-1}b}{1 + \gamma e^\top T^{-1}e} T^{-1}e,$$

which requires just two cheap tridiagonal solves, $T^{-1}b$ and $T^{-1}e$.

¹⁰ For the sake of simplicity, we use the plain `\` factorizes A from scratch in every step here. A KLU that reused the symbolic factorization was recorded around 5 ms for $n = 8000$.

Table 1: Time per Newton step for the heating problem, including tracing the program, building A , and solving with it by `\` (best of three runs).

Example 4: The heating problem in JAX

¹¹ The same holds for the backends of `DifferentiationInterface.jl`, whose sparse mode does not help when J is dense; Section 3 shows how lifting the program makes it help.

n	jax.jacfwd		jax.jacrev	
	Prepare	Step	Prepare	Step
500	70 ms	1.2 ms	71 ms	1.2 ms
1000	72 ms	4.5 ms	72 ms	4.0 ms
2000	0.10 s	34 ms	98 ms	33 ms
4000	0.29 s	0.21 s	0.28 s	0.21 s
8000	1.7 s	1.4 s	1.9 s	1.5 s

The mode makes no difference, since J is square. Due to J being dense things scales poorly and at $n = 16000$ things takes around 17s.

Table 2: Time per Newton step for the heating problem in JAX, in double precision, including forming J and solving with it by `jnp.linalg.solve` (best of three runs). The preparation is the first call, which compiles the step.

3. Lifting

This could seem like the *end* for using standard AD software for getting the sparse augmented Jacobian, however that is not true. A simple technique called *Lifting* [5] instead rewrites the program, so that the intermediate variables become unknowns. That is, the lifted program returns both u and $f(x)$, and the forward relation and the outputs become equations. Said differently: The lifted program has a Jacobian that is exactly the augmented matrix A of Equation 11¹²

$$G(u, x) = \begin{bmatrix} u \\ Pu \end{bmatrix} - \begin{bmatrix} \Phi(u, x) \\ 0 \end{bmatrix}, \quad G(u(x), x) = \begin{bmatrix} 0 \\ f(x) \end{bmatrix},$$

$$\frac{\partial G}{\partial(u, x)} = \begin{bmatrix} I - \frac{\partial \Phi}{\partial u} & -\frac{\partial \Phi}{\partial x} \\ P & 0 \end{bmatrix} = \begin{bmatrix} L & -M^\top \\ P & 0 \end{bmatrix}, \quad (15)$$

Now the Jacobian of G is sparse, even when L^{-1} is dense. That means that tools such that *DifferentiationInterface.jl* or *adsex* can be used to extract the sparse Jacobian. By hand, lifting every operation would mean rewriting the whole program as equations, so in practice only the few intermediate variables that couple many others are lifted, such as the integral in the heating problem.

Example 5 (Lifting the heating problem by hand) We lift the partial sums of the integral, $s_k = s_{k-1} + \exp(-x_k)$ with $s_0 = 0$, rather than only the total, whose equation would read every input, and the outputs $r = F(x, s_n)$, where F is the heating residual of Example 3 with $s = s_n$. The lifted variables are $u = (s, r)$, the outputs are picked out by $P = [0 \ I]$, and the lifted program is

$$G(s, r, x) = \begin{bmatrix} Ls - e(x) \\ r - F(x, s_n) \\ r \end{bmatrix}, \quad \frac{\partial G}{\partial(s, r, x)} = \begin{bmatrix} L & 0 & \text{diag}(e) \\ -\frac{\partial F}{\partial s} & I & -\frac{\partial F}{\partial x} \\ 0 & I & 0 \end{bmatrix}, \quad (16)$$

where L is the bidiagonal matrix of Example 1 and $e_k = \exp(-x_k)$. The first block row is the cumulative sum of Example 1, the second defines the outputs, and the third sets them to zero, so the Jacobian has exactly the form of A in Equation 11.¹³

¹² $G(u, x) = (0, f(x))$ holds only at $u = u(x)$, the values the program computes; for other u , the first block measures how far u is from satisfying the forward relation.

Example 5: Lifting the heating problem by hand

¹³ The last block row says $r = 0$, so r can be eliminated, which leaves $G(s, x) = \begin{bmatrix} Ls - e(x) \\ F(x, s_n) \end{bmatrix}$ with n fewer unknowns. The scripts and Table 3 use this smaller form.

n	$\text{nnz}(\partial G)$	DifferentiationInterface.jl		asdex	
		Prepare	Step	Prepare	Step
500	3 497	0.22 ms	0.34 ms	57 ms	0.79 ms
1000	6 997	0.41 ms	0.64 ms	38 ms	0.96 ms
2000	13 997	0.87 ms	1.3 ms	77 ms	4.2 ms
4000	27 997	2.3 ms	3.1 ms	0.12 s	3.1 ms
8000	55 997	24 ms	9.7 ms	0.25 s	4.0 ms

Table 3: Newton steps for the lifted heating problem. The preparation detects the pattern of ∂G and groups its columns, and for asdex also compiles the Jacobian; it is timed once. A step computes ∂G and solves with it, with \ for DifferentiationInterface.jl and with SciPy’s sparse LU for asdex, since JAX has no sparse direct solver; best of three runs.

Example 6: Estimating the heating parameter

Example 6 (Estimating the heating parameter) The same idea regarding lifting can be used in mathematical programming software like JuMP.jl. In this example try to estimate λ based on a set of noisy temperature measurements, using the UnoSolver, that is solving the following nonlinear program

$$\begin{aligned} \min_{x, \lambda \geq 0} \quad & \sum_{i=1}^n (x_i - y_i)^2 \\ \text{subject to} \quad & x_{i-1} - 2x_i + x_{i+1} + \frac{\lambda \exp(-x_i)}{s^2} = 0, \quad i = 1, \dots, n. \end{aligned} \quad (17)$$

In JuMP.jl, the sum $s = \sum_j \exp(-x_j)$ can be written in two ways. Directly defining s as an expression

```
s = @expression(model, sum(exp(-x[j]) for j in 1:n))
```

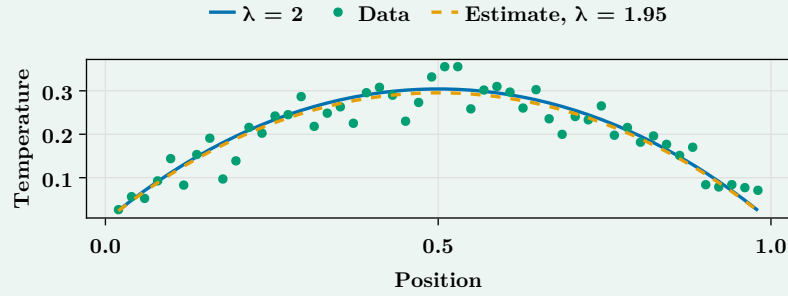
and lifted, it is a variable with a constraint of its own,

```
@variable(model, s, start = n) # sum(exp(-x[j])) = n at the start x = 0
@constraint(model, s == sum(exp(-x[j]) for j in 1:n))
```

The expression is not a fixed number but a shorthand, which JuMP.jl substitutes into every constraint that uses s . As a variable, s is an unknown of its own, which Uno updates along with x , and its constraint makes it equal the sum only at the solution. With the expression, every constraint reads every x_j , so the Jacobian and the Hessian that JuMP.jl hands to Uno become dense. Lifted, only the constraint of s reads all of x , and since JuMP.jl differentiates each constraint on its own, that one full row is cheap and both matrices are sparse.

n	Lifted		Expression	
	Time	Memory	Time	Memory
100	5 ms	2.6 MiB	3.0 s	809 MiB
200	28 ms	5.2 MiB	10.6 s	4.2 GiB
1000	36 ms	25 MiB	—	—
2000	140 ms	50 MiB	—	—
4000	192 ms	102 MiB	—	—

Table 4: Solving the estimation with s lifted and as an expression, with Uno through JuMP.jl; both estimate $\lambda \approx 2.00$. The memory is that allocated in Julia.



The script is `heating_lifted.jl`.

Figure 1: The estimation for $n = 50$ with the true $\lambda = 2$. The data have noise with standard deviation 0.03 added.

4. Summary

Every Jacobian of a program is the product $J = PL^{-1}M^\top$. Even when the three matrices are sparse, the inverse L^{-1} can be dense, resulting in a dense Jacobian. Sparse differentiation pays off when J is sparse, but not when it is dense. In this case actually knowing the structure of the Jacobian can be useful. Either directly knowing P , L , and M gives us a sparse representation of the Jacobian, or lifting the program to expose intermediate variables gives a sparse Jacobian which can be connected to the original dense Jacobian through a Schur complement.¹⁴

Bibliography

- [1] A. Wächter and L. T. Biegler, “On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming,” *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006, doi: 10.1007/s10107-004-0559-y.
- [2] C. Vanaret and S. Leyffer, “Implementing a Unified Solver for Nonlinearly Constrained Optimization,” *Mathematical Programming Computation*, 2026, doi: 10.1007/s12532-026-00310-9.
- [3] A. Hill, G. Dalle, and A. Montoison, “An Illustrated Guide to Automatic Sparse Differentiation.” [Online]. Available: <https://iclr-blogposts.github.io/2025/blog/sparse-autodiff/>
- [4] A. A. Lacey, “Thermal Runaway in a Non-Local Problem Modelling Ohmic Heating. Part I: Model Derivation and Some Special Cases,” *European Journal of Applied Mathematics*, vol. 6, no. 2, pp. 127–144, 1995.
- [5] J. Albersmeyer and M. Diehl, “The Lifted Newton Method and Its Application in Optimization,” *SIAM Journal on Optimization*, vol. 20, no. 3, pp. 1655–1684, 2010.

¹⁴ Eliminating the intermediate variables from the lifted Jacobian leaves the Schur complement of the upper-left block L in $\begin{bmatrix} L & -M^\top \\ P & 0 \end{bmatrix}$, which is $0 - PL^{-1}(-M^\top) = PL^{-1}M^\top = J$.