A SURVEY OF AUTOMATIC DIFFERENTATION TECHNIQUES FOR NONLINEAR NODAL ANALYSIS

HOLLY M. JACKSON*

1. Introduction. Many complex nonlinear systems can be modeled using nodal analysis. Due to their grid-like construction, nodal systems often have sparse Jacobian matrices, and we can take advantage of this sparsity to accelerate their computation. An implementation of sparse forward-mode or reverse-mode automatic differentiation can be used to compute the Jacobian in a single call.

However, when a node in the system has a disproportionately high degree of connectivity, the Jacobian may be mostly sparse, with several dense rows. In these cases, it is more difficult to apply sparse automatic differentiation techniques.

In this paper, I present a basic formulation of a nodal system and implement forward-mode and reverse-mode automatic differentiation to construct its Jacobian. I implement a matrix coloring scheme to accelerate sparse forward-mode automatic differentiation of the Jacobian of nodal systems. In addition, I implement a combined sparse automatic differentiation technique to compute a Jacobian that is mostly sparse, with several dense rows. I demonstrate my techniques on several examples of ranging complexity. I solve these systems using Newton's method computing the Jacobian with the most efficient of my methods at each iteration. Finally, I perform a thorough performance analysis on my methods and compare them to existing libraries in Julia.

2. System formulation and matrix construction. In this section, I present a general method to formulate a nonlinear nodal system defined by nodes, edges, and initial conditions, based on [1]. A nodal system of this nature could represent a wide variety of systems, such as lattices, resistive networks, economic models, and heat transfer. Any system modeled on node-to-node relationships can be adapted to this model.

A simple example of a nodal system – a small lattice – is shown in Figure 1. The lattice has six unfixed nodes and fourteen edges. A initial load force f_L is acting on node 3.

Let us formulate a nonlinear system $f(u)$ that calculates the force on each node given node displacements u. I construct this system using two relationships: the conservation equations and constitutive equations. This same framework can be applied to other nodal systems using the correct physical relationships.

Let us define two initial unknown quantities in our system: the forces on each strut f_s and the displacements of each node u. It is important to remember each node is a 2D position $n = (x, y)$ and each force is a 2D vector $f = (f_x, f_y)$.

The law of conservation tells us the sum of the forces at each node should be 0 (or the load force on that node) in each direction x and y. For our simple example, this would mean for example, for node 1 $f_{1,x} + f_{2,x} + f_{3,x} = 0$ and $f_{1,y} + f_{2,y} + f_{3,y} = 0$. For node 3, experiencing a load force f_L, we have the following conservation equations: $f_{4,x} = -f_{L,x}$ and $f_{4,y} = -f_{L,y}$.

I can represent these relationships for the whole system using an incidence matrix A such that $Af_s = f_L$.

*MIT EECS, Undergraduate Class of 2022 (hjackson@mit.edu).

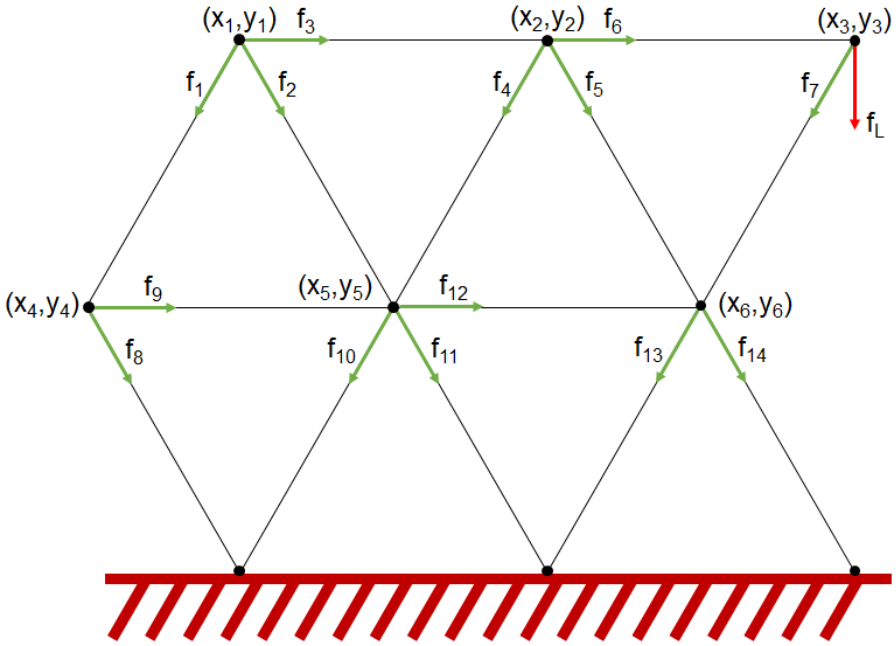


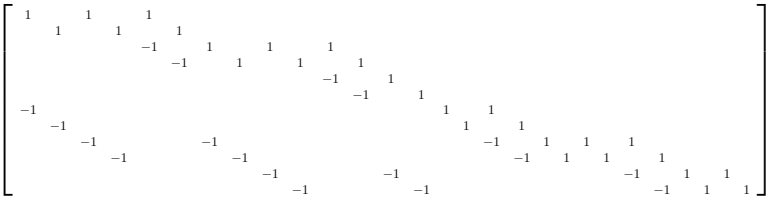
FIG. 1. *A simple nonlinear lattice system. Forces through each strut labeled in green. An external load f_L is acting on node 3 at (x_3, y_3).*

$$A \begin{bmatrix} f_{1,x} \\ f_{1,y} \\ f_{2,x} \\ f_{2,y} \\ \vdots \\ f_{27,x} \\ f_{27,y} \\ f_{28,x} \\ f_{28,y} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -f_{load,x} \\ -f_{load,y} \\ 0 \\ 0 \\ \vdots \end{bmatrix} \tag{2.1}$$

The pseudocode in Algorithm 2.1 shows a general node-stamping method to construct A for a lattice system with N vertices and S struts.

The equation below shows the incidence matrix for the simple system in Figure 1. Notice A is a sparse matrix, and is even sparser for more complex systems like those discussed in Section 4.

$$A = \tag{2.2}$$



Algorithm 2.1 Generating the incidence matrix

```
Define A, a 2N × 2S
Define f_L, a 2N-length vector
for each strut s_i with force f_i from (n_1, n_2) do
  if n_1 is not fixed then
    A(n_{1,x}, s_{i,x}) = 1
    A(n_{1,y}, s_{i,y}) = 1
  end if
  if n_2 is not fixed then
    A(n_{2,x}, s_{i,x}) = −1
    A(n_{2,y}, s_{i,y}) = −1
  end if
end for
for each load f_{load,i} applied to node n_i do
  f_L(n_{i,x}) = −f_{load,i,x}
  f_L(n_{i,y}) = −f_{load,i,y}
end for
return A, f_L
```

The constitutive equations for our system relate the forces through each strut to the node displacements. I can calculate the force through a given strut from some node n_1 with initial position (x_1, y_1) to some node n_2 with initial position (x_2, y_2) according to the following equations:

$$\begin{aligned} f_x &= \epsilon \frac{(x_2 + \Delta x_2) - (x_1 + \Delta x_1)}{L}(L_0 - L) \\ f_y &= \epsilon \frac{(y_2 + \Delta y_2) - (y_1 + \Delta y_1)}{L}(L_0 - L) \end{aligned} \tag{2.3}$$

where L_0 is the nominal length of the strut and L is the length of the strut after displacement

$$\begin{aligned} L &= \sqrt{((x_2 + \Delta x_2) - (x_1 + \Delta x_1))^2 + ((y_2 + \Delta y_2) - (y_1 + \Delta y_1))^2} \\ L_0 &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1^2)} \end{aligned} \tag{2.4}$$

and ϵ is a parameter representing the material stiffness of the strut.

The constitutive equations are nonlinear in the node displacements.

Let us define a vector v to hold the original node positions of the lattice.

$$v = [x_1, y_1, x_2, y_2, \ldots, x_N, y_N]^T$$

I also define a vector

$$s_{fix} = [0, 0, 0, 0, \ldots, -x_{fixed}, -y_{fixed}, \ldots]^T$$

of length $2S$ which for each strut contains the x and y positions of a fixed vertex if the strut is connected to one.

Finally, I define a sparse matrix α of size $2S \times 2S$, which for each strut s_i is one at $(2i, 2i)$, $(2i+1, 2i)$, $(2i, 2i+1)$, and $(2i+1, 2i+1)$ (see sparsity pattern in Figure

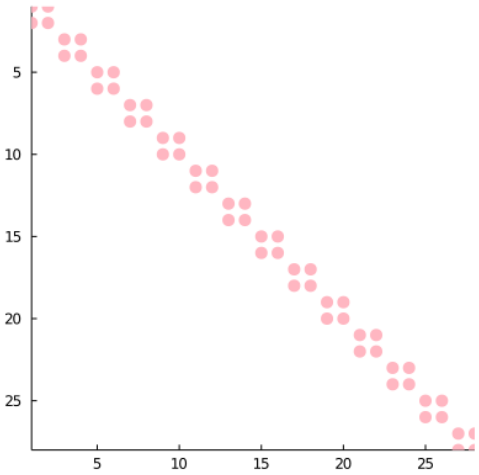


FIG. 2. *Sparsity plot of α matrix for simple system in Figure 1.*

2). This allows us to quickly calculate the strut lengths to use in the constitutive equations with sparse matrix operations.

Given these quantities, I can define L_0 and L in linear algebraic form.

$$\begin{aligned} L_0 &= (\alpha(A^T v + s_{fix}).^2).^{1/2} \\ L &= (\alpha(A^T(u+v) + s_{fix}).^2).^{1/2} \end{aligned} \tag{2.5}$$

Similarly, I can construct the constitutive equation in linear algebraic form.

$$f_s = (\epsilon(A^T(u+v) + s_{fix})./L. * (L_0 - L)) \tag{2.6}$$

Now, to formulate $f(u)$, I combine our conservation and constitutive equations as follows.

$$\begin{aligned} f_s - (\epsilon(A^T(u+v) + s_{fix})./L. * (L_0 - L)) &= 0 \\ A \cdot (f_s - (\epsilon(A^T(u+v) + s_{fix})./L. * (L_0 - L))) &= A \cdot 0 \\ f_L - \epsilon A(A^T(u+v) + s_{fix})./L. * (L_0 - L)) &= 0 \\ \epsilon A(A^T(u+v) + s_{fix})./L. * (L_0 - L)) &= f_L \end{aligned} \tag{2.7}$$

This gives us our final system $f(u) = \epsilon A(A^T(u+v) + s_{fix})./L. * (L_0 - L)) - f_L$.

3. Numerical methods. In order to analyze nodal systems like those outlined in the previous section, I use a variety of automatic differentiation methods to compute the Jacobian matrix. The Jacobian matrix describes the sensitivities of each node position as the function $f(u)$ changes.

I implemented different automatic differentiation approaches. The performance of each method depends on the dimensions and sparsity pattern of the Jacobian matrix and is analyzed thoroughly in Section 6.

First, I implemented forward-mode automatic differentiation [4], and accelerated it for sparse matrices using my own implementation of greedy matrix coloring (Sec 3.1) [3].

Second, I implemented reverse-mode automatic differentiation, by deriving the pullback function $\mathcal{B}_u^f(\lambda)$ of the nodal system [2].

Finally, I merged the the two techniques in a combined sparse automatic differentiation method which applies reverse-mode AD to compute dense rows of the Jacobian while using sparse forward-mode AD (accelerated by matrix coloring) to compute the rest of the Jacobian.

I applied my various automatic differentiation techniques in a an iterative Newton solver. Newton's method needs to recompute the Jacobian matrix at each new guess of the solution, and as such was a good way test the accelerated differentiation techniques. I implemented Newton's method to solve several example nodal systems (see Sec 4 and 5).

3.1. Forward-Mode Automatic Differentiation. We can compute the Jacobian of our system in a single function call using forward-mode automatic differentiation.

Let us define a multi-dimensional dual number using the array of structs representation

$$D = D_0 + \Sigma\epsilon \tag{3.1}$$

where the sensitivity of the function is propagated along the rows of the matrix Σ.

We can compute the derivative $f'(D_0)$ in all directions of Σ in one call of f. $f'(D_0)$ is the Jacobian of our system.

$$f(D) = f(D_0) + f'(D_0)\Sigma\epsilon \tag{3.2}$$

I implemented forward-mode automatic differentiation for a general nodal system of the form defined in Section 2 for a multidimensional dual number. I checked my implementation for correctness against Julia's `ForwardDiff.jacobian`.

3.1.1. Sparse Forward-Mode AD using Matrix Coloring. When the Jacobian is sparse, we can use matrix coloring to accelerate its computation. If we can identify which columns are independent (i.e. their ϵ terms would not collide when computed at the same time), we can compute them simultaneously while still getting the correct value for every row of the Jacobian.

We can find these relationships using graph coloring. If we know the sparsity pattern of the Jacobian, we can build a graph of the column connectivity of the Jacobian. We can apply a distance-1 coloring method to the graph to guarantee no two adjacent nodes are the same color. All columns that are the same color can be computed simultaneously. We can then decompress the final output to recover the full Jacobian matrix.

Figure 3 shows the Jacobian matrix for the simple system in Figure 1 and its compressed representation.

I implemented matrix coloring for a general nodal system of the form defined in Section 2. I verified that the decompressed Jacobian still matched the result without matrix coloring.

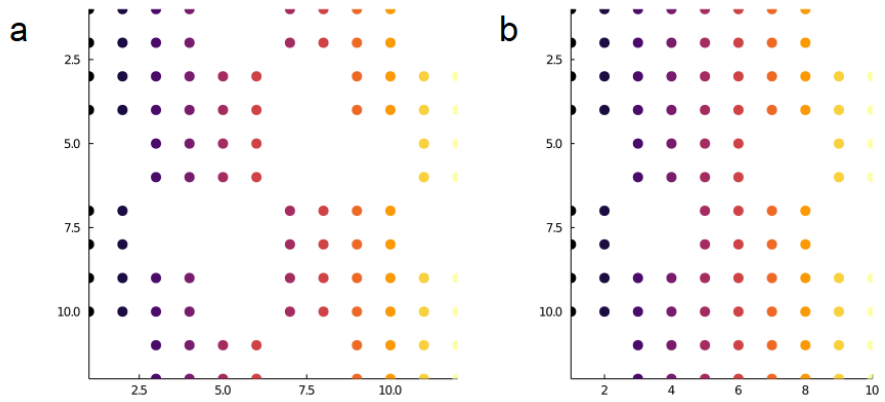


FIG. 3. *(a) The Jacobian matrix for the simple system in Fig 1 and (b) its compressed representation.*

3.2. Reverse-Mode Automatic Differentiation. When the size of the input of our system is smaller than the size of the output of our system, it may be faster to use reverse-mode automatic differentiation to compute the Jacobian of a system. Reverse-mode AD computes the Jacobian row-by-row, while forward-mode AD computes the Jacobian column-by-column.

To break down the steps of reverse accumulation, we can rewrite $f(u)$ as composed functions $f = f^L \circ f^{L-1} \circ \cdots \circ f_1$.

$$\begin{aligned} f &= \epsilon A w - f_L \\ w &= z./y.^{1/2}.*(L_0 - y^{1/2}) \\ y &= \alpha x \\ x &= z.^2 \\ z &= A^T(u+v) + s_{fix} \end{aligned} \tag{3.3}$$

We can formulate reverse-mode AD through a successive application of pullback functions going in the reverse direction on these composed functions.

$$\mathcal{B}^u_f(A) = B^u_{f^1}\left(\ldots\left(\mathcal{B}^{f^{L-2}(f^{L-3}(\ldots f^1(u)\ldots))}_{f^{L-1}}\left(\mathcal{B}^{f^{L-1}(\ldots f^{L-2}(\ldots f^1(x)\ldots))}_{f^L}(A)\right)\right)\ldots\right)$$

We can calculate the derivatives with respect to each quantity in reverse order, pulling back on the vector λ.

$$\begin{aligned} \bar{f} &= \lambda = [1,1,1,1,\ldots]^T \\ \bar{w} &= \epsilon A^T \lambda \\ \bar{y} &= \bar{w}.*\left(-z.*\frac{1}{2}y.^{-3/2}.*\left(L_0 - y.^{1/2}\right) + z./y.^{1/2}.*\left(-\frac{1}{2}y.^{-1/2}\right)\right) \\ \bar{x} &= \alpha^T \bar{y} \\ \bar{z} &= \bar{x}.*2z + \bar{w}.*(1./y.^{1/2}.*(L_0 - y.^{1/2})) \\ \bar{u} &= A\bar{z} \end{aligned} \tag{3.4}$$

We can compute a single row of the Jacobian by pulling back on a basis vector for row of Jacobian you want to compute $\mathcal{B}_f^u(v_i)$. Instead, the whole Jacobian can be computed in one step by pulling back on the identity matrix, $J = \mathcal{B}_f^u(I)$. I checked my implementation for correctness against Julia's `ForwardDiff.jacobian`.

3.2.1. Sparse Reverse-Mode AD using Matrix Coloring. Similarly to forward-mode AD, matrix coloring can be applied to speed up sparse reverse-mode AD. This simply requires assigning colors to rows in the graphs rather than columns. I implemented matrix coloring to accelerate reverse-mode automatic differentiation as well. I verified my implementation against Julia's `ForwardDiff.jacobian`.

3.3. Combined Sparse Automatic Differentiation. .

In an irregular system where some rows are dense but much of the Jacobian remains sparse, neither forward-mode nor reverse-mode automatic differentiation optimally accelerate the computation of the Jacobian. To handle Jacobians with irregular sparsity patterns, we can combine reverse-mode AD and sparse forward-mode AD to accelerate the computation.

First, we can compute the sparsity pattern S of the Jacobian a single time with one forward (or reverse) pass through the system (or using `SparsityDetection.jl`). Using a heuristic to determine the density of the rows of the Jacobian from the sparsity pattern (e.g. $> 50\%$ of row nonzero), we can select a subset of rows to be computed using reverse-mode automatic differentiation. We then color the rest of the graph (ignoring these dense rows) and perform sparse forward-mode automatic differentiation. Finally, we decompress the Jacobian and merge the results. This will decrease the total number of columns being computed in the forward pass, accelerating the differentiation overall if the overhead of the reverse mode is low enough.

Combined sparse automatic differentiation can also be applied in reverse. We can instead compute dense columns using forward-mode AD, and then compute the rest of the Jacobian row-wise using reverse-mode automatic differentiation accelerated with matrix coloring (ignoring columns already computed by the forward-mode AD).

I implemented both types of combined sparse automatic differentiation. As a proof of concept, I ran both implementations on the simple example. I checked my implementation for correctness by against Julia's `ForwardDiff.jacobian`.

3.4. Linear Solving. We can use Newton's method to solve our nonlinear equation, now that we have a quick method to compute the Jacobian using forward-mode, reverse-mode, or combined sparse automatic differentiation, depending on the size and sparsity pattern of the Jacobian. Starting from an initial guess u_0, we can use Newton's method to iteratively follow the tangent line of f until it nears the root of the function.

At each iterative step, we update our guess for the root $u*$ such that gives $f(u*) = 0$ using the following relation.

$$u_{k+1} = u_k - J(u_k)^{-1} f(u_k) \tag{3.5}$$

We can solve this numerically in two stages:

1. Solve $Ja = f(u_k)$ for a
2. Update $u_{k+1} = u_k - a$

We perform the linear solve in step 1 by first factorizing the Jacobian and then using sparse LU decomposition, which is more efficient for sparse systems than computing the inverse of the Jacobian or using Julia's backslash operator. The pseudocode

for my implementation of Newton's method on my system is shown in Algorithm 3.1.

Algorithm 3.1 Newton's method

function `newton_step`(f, u_0):
  $J =$ `forward_mode_AD`(f, u) / `reverse_mode_AD`(f, u) / `combined_mode_AD`(f, u)
  Factorize J
  Use sparse LU decomposition to solve $Jd = f(u_0)$ with the factorized Jacobian
  return $u_0 - d$
function `newton`(f, u_0):
  $u_{last} = u_0$
  for $i = 1 :$ max_iter **do**
    $u =$ `newton_step`(f, u_{last})
    if $(||f(u)||_2 < 10^{-12}) \cap (||u - u_{last}||_2 < 10^{-12})$ **then**
      return u
    end if
    $u_{last} = u$
  end for
  return u

Figure 4 shows the equilibrium state of the simple system from Figure 1, computed using Newton's method.

4. Example system 1: a complex lattice. Let us use automatic differentiation to solve a large complex lattice system as an example. The blue lattice in Figure 6 shows a lattice with 5000 nodes, structured similarly to the simple example in Figure 1. I can formulate this system using the nodal stamping approach outlined in Section 2. Similarly to simple example, I apply a downwards force (2.5 units) to the node in upper right corner of lattice.

Since each node is only dependent on directly neighboring nodes, the Jacobian of the system is very sparse as can be seen in Figure 5. The Jacobian matrix, originally of size 10000×10000, can be compressed to 10000×20 using distance-one matrix coloring; the figure shows each column labeled with its respective color.

Using the compressed Jacobian and forward-mode (or reverse-mode) automatic differentiation, we can accelerate the solving of the original lattice system using Newton's method. Figure 6 shows the original lattice system and the displaced result side by side. Combined sparse automatic differentiation is not necessary to accelerate this example since the Jacobian does not have irregular sparsity,

5. Example system 2: a nodal system with an irregular sparsity pattern. Not all nodal systems show a regular sparsity pattern like in example system 1. For example, take a resistive network where many of the nodes are connected to ground, or a hanging truss where many modes are connected to a single pin point. These nodes are mutually dependent on many of the other nodes in the systems, and this will add density to the Jacobian matrix.

To test combined sparse automatic differentation, I construct an example with a Jacobian with an irregular sparsity pattern. I can do this by adding several high-degree nodes to the structure (i.e. nodes connected to many other nodes in the structure). We define the system to have 380 nodes and 2030 edges. The effect of this is clearly visible in the Jacobian in Figure 7. While most of the Jacobian is sparse, there are several very dense rows. This prevents us from easily applying sparse

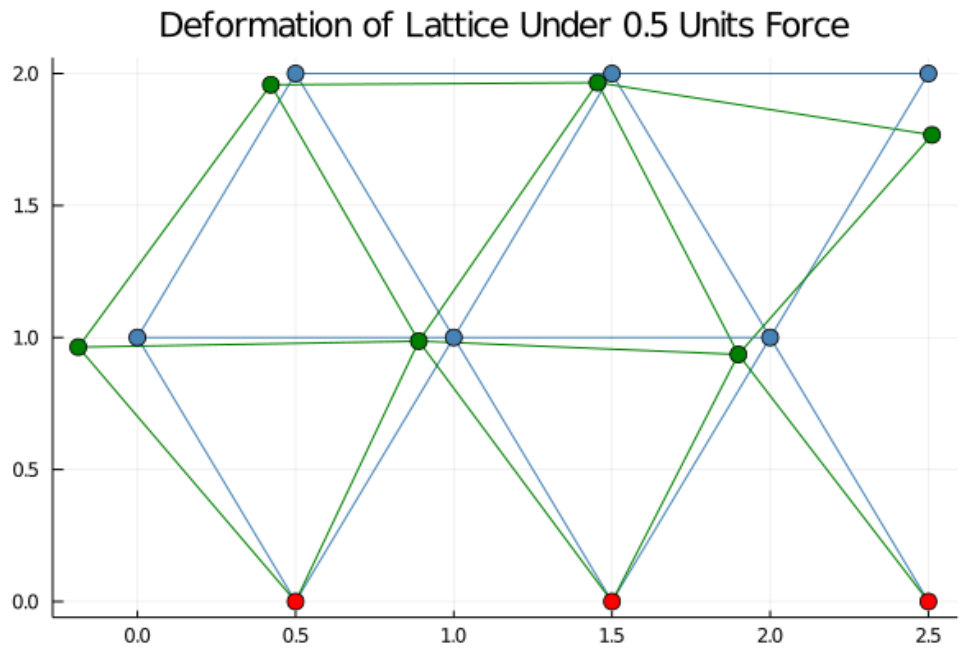


FIG. 4. *Solution of the simple system from Figure 1 computed using Newton's method. Original node and strut positions shown in blue. Given a load force* $f_L = 0.5$ *in the negative y-direction applied to node 3, the final displaced node and strut positions are shown in green. Fixed nodes marked red.*

differentiation techniques.

To overcome this and still accelerate the automatic differentiation, we can use the combined automatic differentiation technique presented in Section 3.3. There are two possible approaches. I can identify the six dense rows in the Jacobian matrix (rows with over 50% nonzero elements), and compute these in one pass using reverse-mode automatic differentiation. Then, using matrix coloring, I can condense the 760×760 Jacobian matrix into a 760×26 compressed representation and use forward-mode automatic differentiation to accelerate the rest of the computation. Merging the result of the forward-mode AD and reverse-mode AD will reconstruct the entire matrix.

Conversely, I can identify the six dense columns in the Jacobian matrix and compute these with forward-mode AD. Then, I would use sparse reverse-mode AD with matrix coloring to compute the rest of the Jacobian. I implemented both and compared their performance in the following section.

Using either of these two methods to compute the Jacobian at each iteration, I run Newton's method to the original lattice system. Figure 8 shows the original lattice system and the displaced result side by side. The high-degree nodes are clearly visible in the structure, adding significant additional support. After a 2 unit load force is applied to the node in the top right corner, the structure displaces only slightly, reinforced by its rigid internal structure.

6. Performance Analysis. I analyzed the performance of my automatic differentiation methods for each example presented in this paper, since the performance depends on the size and sparsity pattern of the Jacobian of the specific system. In

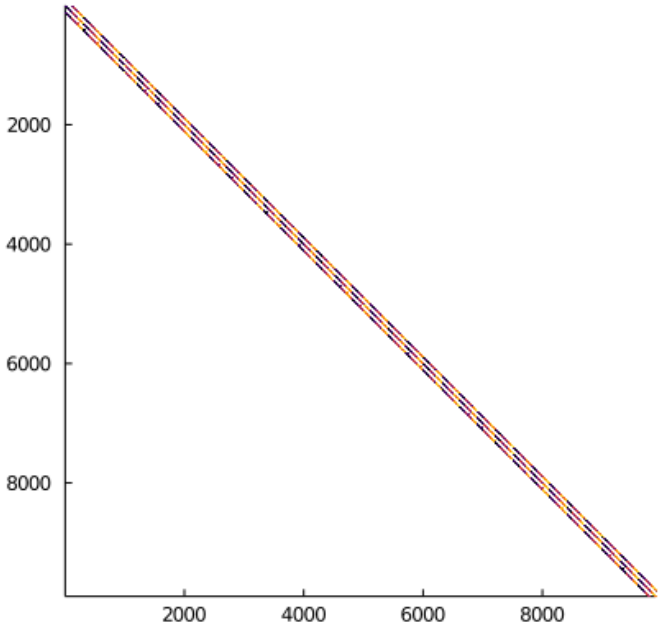


FIG. 5. *Sparse* 10000×10000 *Jacobian matrix for example system 1, colored with 20 colors. The matrix can be compressed to* 10000×20.

addition, I compared my implementations to existing Julia libraries.

In general, my implementation of forward-mode AD was less efficient than standard Julia libraries. However, my implementation of reverse-mode AD not only fared better than standard Julia reverse-mode differentiation libraries, it also even fared better than Julia's `ForwardDiff.jacobian` for large systems. My reverse-mode AD even fared better than `ForwardDiff` *without* acceleration using matrix coloring for some examples.

For all examples, sparse vectors and matrices were represented using Julia's `SparseArrays.jl` library.

6.1. Simple example. Let us first analyze the performance of the simple system. Since the simple system is a small lattice, sparse differentiation methods are not necessary to accelerate the computation of the Jacobian and may add unnecessary overhead, causing these methods to be less efficient on this example. As a result, we would expect forward-mode or reverse-mode AD to be the most optimal differentiation method for the simple system. For reference, a forward pass through $f(u)$ for the simple system takes $6.280\mu s$ (72 allocations: 10.94 KiB).

Table 3 shows a thorough performance analysis of the simple system using my implementations of forward-mode and reverse-mode automatic differentiation, compared to Julia's existing `ForwardDiff.jl` and `ReverseDiff.jl` differentiation packages. My implementation of reverse-mode automatic differentiation was able to compute the Jacobian the fastest of all methods, supporting our intuition.

The table shows that my implementation of forward-mode AD is less efficient than Julia's `ForwardDiff.jl`. However, my implementation of reverse-mode is more

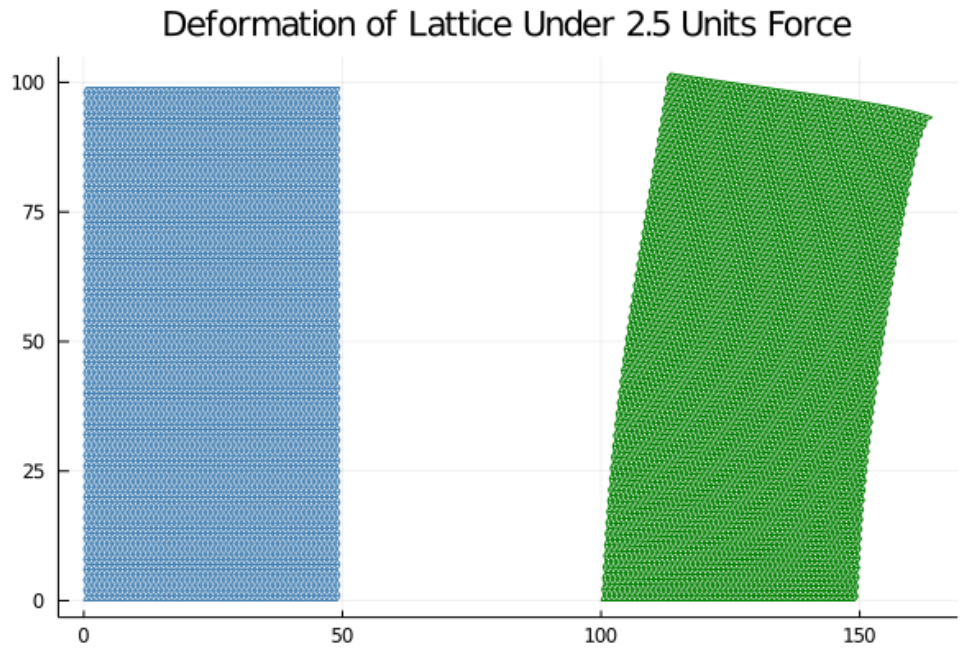


FIG. 6. *Solution to example system 1 computed using Newton's method. Original node and strut positions shown in blue in the lattice on the left. Given a load force* $f_L = 2.5$ *in the negative y-direction applied to the top right node, the final displaced node and strut positions are shown in green in the lattice on the right.*

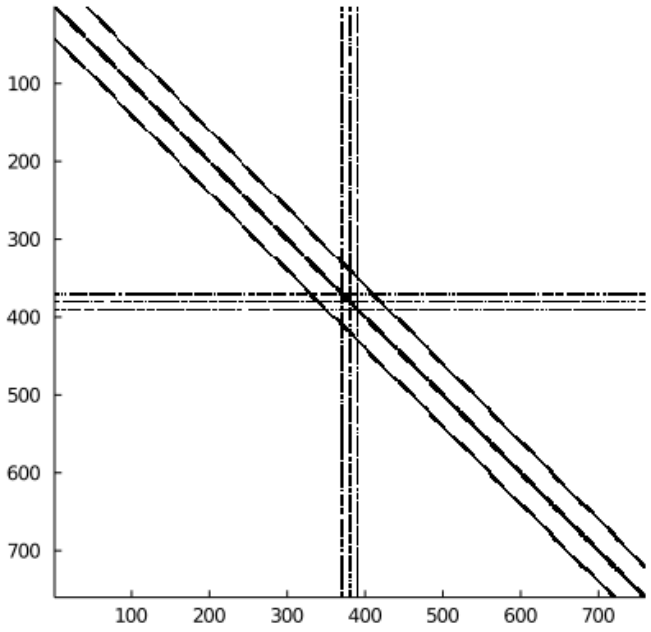


FIG. 7. *Sparsity pattern of the Jacobian matrix for example system 2.*

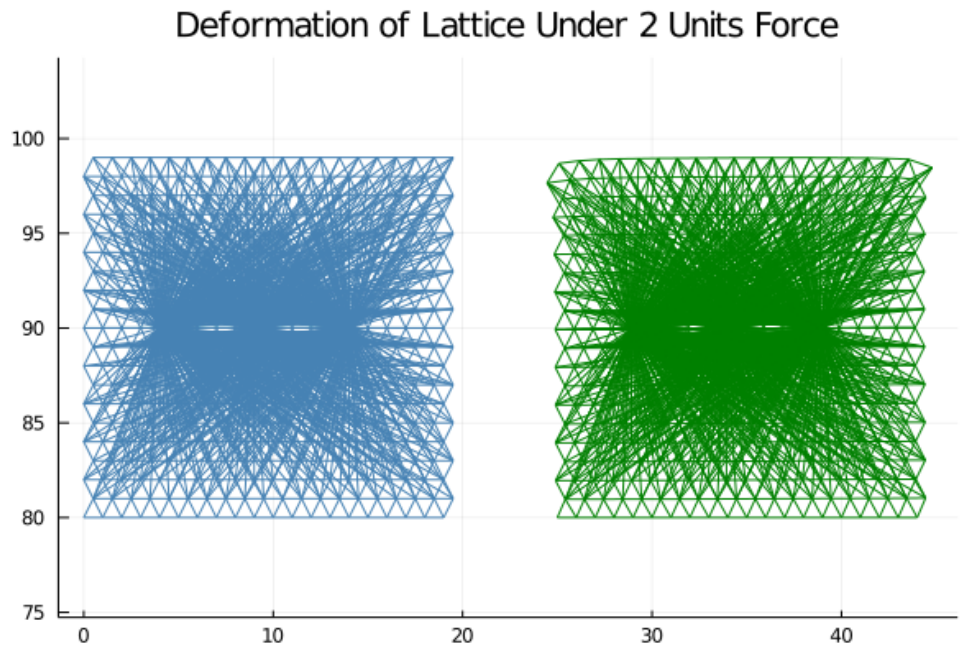


FIG. 8. *Solution to example system 2 computed using Newton's method. Original node and strut positions shown in blue in the lattice on the left. Given a load force $f_L = 2$ in the negative y-direction applied to the top right node, the final displaced node and strut positions are shown in green in the lattice on the right.*

Method	**Performance**		
	Time (μs)	# Alloc.	Memory (KiB)
Julia `ForwardDiff.jacobian`	18.500	83	65.97
My `forward_mode_AD`	121.401	169	51.39
`forward_mode_AD` + matrix coloring	159.499	669	103.94
Julia `ReverseDiff.jacobian`	244.899	327	54.38
My `reverse_mode_AD`	35.101	158	51.66
`reverse_mode_AD` + matrix coloring	90.899	648	103.72

TABLE 1
Performance analysis of the simple system. Peak performance in each category is colored green.

efficient that Julia's `ReverseDiff.jl`.

Although I tested sparse forward-mode differentiation and combined sparse differentiation as a proof-of-concept on the simple system, they performed less efficiently because the simple system does not have a particularly sparse or irregular pattern.

6.2. Example system 1. Example system 1 has a sparse Jacobian matrix, as shown in Figure 5. As a result, we would expect matrix coloring to help accelerate the computation of the Jacobian matrix.

Table 3 shows a thorough performance analysis of this example using my implementations of forward-mode and reverse-mode automatic differentiation, compared to Julia's existing `ForwardDiff.jl` and `ReverseDiff.jl` differentiation packages. In

Method	Performance		
	Time (s)	# Alloc.	Memory (GiB)
Julia `ForwardDiff.jacobian`	32.637	87428	38.95
My `forward_mode_AD`	78.772	220	21.70
`forward_mode_AD` + matrix coloring	17.115	532095	24.04
Julia `ReverseDiff.jacobian`	> 600	–	–
My `reverse_mode_AD`	7.488	229	17.36
`reverse_mode_AD` + matrix coloring	16.916	532087	24.04

TABLE 2
Performance analysis of example system 1. Peak performance in each category is colored green.

addition, I benchmark my implementation of forward-mode automatic differentiation accelerated with matrix coloring.

The complex system has 5000 nodes and a 10000×10000 Jacobian matrix. A forward pass through $f(u)$ for the complex system takes 1.830 ms, with 108 allocations using 6.22 MiB of memory.

As can be seen in the table, forward-mode AD with matrix coloring significantly accelerates the computation of the Jacobian ($> 4\times$ my forward-mode alone, and almost $2\times$ the native Julia implementation), even using a less optimal implementation of forward-mode automatic differentiation.

However, reverse-mode AD fares better than reverse-mode AD with matrix coloring for this example system. This is likely because of overhead added by decompressing the Jacobian.

It is important to note, there is an overhead to compute the coloring from the sparsity pattern, but is required only once and can be reused each sequential computation. In my implementation, computing the sparsity pattern takes the same duration as computing one forward or reverse pass of the Jacobian. However, once the sparsity pattern is computed a single time, it can be reused in every Newton iteration without additional overhead.

6.3. Example system 2. Example system 2 has a Jacobian matrix with an irregular sparsity pattern, as shown in Figure 7. Simply using matrix coloring to speed up the computation of the Jacobian will have a limited effect since dense rows in the matrix will limit the amount of compression. As a result, we would expect combined sparse automatic differentiation, in which dense rows (or columns) of the Jaocbian are processed separately, to accelerate the computation of the Jacobian the most for example system 2.

This example has 380 nodes and a 760×760 Jacobian matrix. A forward pass through $f(u)$ for the complex system takes 272.401 μs, with 96 allocations using 964.61 KiB of memory.

Table 3 shows the performance analysis of this example. As can be seen in the table, combined sparse automatic differentiation is the fastest method for this example. Specifically, combined AD fares the best when the dense columns are computed using forward-mode AD and the rest of the matrix is computed using reverse-mode AD accelerated with matrix coloring (which I refer to as dense forward). This method is more than twice as fast as forward-mode AD alone (from Julia's `ForwardDiff.jacobian`).

As expected, forward-mode AD and reverse-mode AD with matrix coloring do not fair much better (if not worse, in the case of reverse-mode AD) than their counterparts without matrix coloring.

Method	Performance		
	Time (ms)	# Alloc.	Memory (MiB)
Julia `ForwardDiff.jacobian`	114.599	6505	412.03
My `forward_mode_AD`	799.475	201	237.44
`forward_mode_AD` + matrix coloring	669.231	35993	423.85
Julia `ReverseDiff.jacobian`	> 6000	–	–
My `reverse_mode_AD`	66.551	209	191.26
`reverse_mode_AD` + matrix coloring	103.958	35989	382.84
My `combined_mode_AD` (dense reverse)	69.030	35999	140.77
My `combined_mode_AD` (dense forward)	53.156	35989	139.37

TABLE 3
Performance analysis of example system 2. Peak performance in each category is colored green.

Additionally, it is interesting to note that my implementation of reverse-mode AD is almost twice as fast as Julia's `ForwardDiff.jacobian` without any acceleration techniques applied.

7. Conclusions. In this paper, I implemented a variety of different automatic differentiation techniques to compute the Jacobian of nonlinear nodal systems. The methods presented here can be quickly adapted to many kinds of nodal systems for a wide variety of applications.

In addition to implementing both sparse forward- and reverse-mode automatic differentiation accelerated by matrix coloring, I also presented a combined automatic differentiation approach that mixed both differentiation techniques to accelerate the computation of Jacobians with irregular sparsity patterns. I developed a particularly efficient implementation of reverse-mode AD for nodal systems, that outperformed Julia's built-in libraries for large systems. However, there is room for improvement to accelerate my implementation of forward-mode AD, which was less efficient that native Julia methods.

Although this paper is a case study of nodal systems, the techniques presented here can be extended to accelerate computation in many different kinds of systems with similar sparsity patterns. The problem of irregular Jacobian sparsity is not unique to nodal systems. In future work, these methods could be applied to other nonlinear problems.

Supplementary note. All supporting code for this project is included in the supplementary file `AD.jl`.

REFERENCES

[1] M. R. Deepak Ramaswamy and K. Veroy, *Equation formulation methods - stamping techniques, nodal versus node-branch form*, Fall 2003.
[2] C. C. Margossian, *A review of automatic differentiation and its efficient implementation*, WIREs Data Mining and Knowledge Discovery, 9 (2019), https://doi.org/10.1002/widm.1305, http://dx.doi.org/10.1002/WIDM.1305.
[3] P. Mishra, *A summer with jacobians*, October 2019.
[4] J. Revels, M. Lubin, and T. Papamarkou, *Forward-mode automatic differentiation in julia*, 2016, https://arxiv.org/abs/1607.07892.