Hierarchical seeding for efficient sparsity pattern recovery in automatic differentiation

Joris Gillis*[1] and Moritz Diehl[2]

[1] KU Leuven, Electrical Engineering Department (ESAT-STADIUS), Kasteelpark Arenberg 10, 3001 Leuven, Belgium
[2]Freiburg Univ, Department of Microsystems Engineering (IMTEK), G.-Koehler-Allee 102, 79110 Freiburg, Germany

Obtaining the Jacobian $J = \frac{\partial f}{\partial x}$ of a vector valued function $f(x) : \mathbb{R}^n \to \mathbb{R}^m$ is central to applications such as gradient-based constrained optimization and sensitivity analysis. When the function is not a black box, but instead an interpretable algorithm with run-time T, automatic differentiation offers a mechanistic way to derive two algorithms from the original $f(x)$ that accurately evaluate the following sensitivities:

Forward sensitivity	$\mathrm{AD}_f^{\mathrm{fwd}}(x, s^{\mathrm{fwd}}) = J(x)s^{\mathrm{fwd}}, \quad s^{\mathrm{fwd}} \in \mathbb{R}^n$
Adjoint/reverse sensitivity	$\mathrm{AD}_f^{\mathrm{adj}}(x, s^{\mathrm{adj}}) = J^T(x)s^{\mathrm{adj}}, \quad s^{\mathrm{adj}} \in \mathbb{R}^m,$

with the run-time of either of the algorithms AD_f a small multiple of T.
A straightforward approach to recover all Jacobian entries is to seed with columns of an identity matrix. In this way, the forward and reverse sensitivities correspond directly to respectively columns and rows of the sought-after Jacobian. For $m \ll n$, the obvious choice is to use m adjoint sensitivities, while in the $n \ll m$ case, using n is cheapest. With this strategy, the cost for a total Jacobian is in the order of $\min(n, m)T$.

If one knows the sparsity of J beforehand, the number of required sensitivities can potentially be drastically reduced. For example, when $n = m$ and J is known to be diagonal, a single sensitivity evaluation with seed $[1, 1, \ldots]^T$ suffices. More generally, a coloring of the *column intersection graph* of the sparsity pattern of J provides a small set of seeds usable to obtain the full Jacobian. We denote such coloring as $col(J)$ and use an existing distance-2 unidirectional algorithm[2].
The potentially dramatic speed-up requires first the sparsity pattern to be obtained. We will assume for the remainder of this work that we can derive the following bitvector-valued dependency functions[3] from the original algorithm f:

Forward dependency	$\mathrm{dep}_f^{\mathrm{fwd}}(d^{\mathrm{fwd}}) \in \mathbb{B}^m, \quad d^{\mathrm{fwd}} \in \mathbb{B}^n$
Adjoint/reverse dependency	$\mathrm{dep}_f^{\mathrm{adj}}(d^{\mathrm{adj}}) \in \mathbb{B}^n, \quad d^{\mathrm{adj}} \in \mathbb{B}^m,$

with $\mathbb{B}$ the Boolean set $\{0, 1\}$. A zero in the dependency function output means that any seed s with sparsity as in the input d, when supplied to the corresponding sensitivity function, would result in a zero sensitivity output in that same location.
A straightforward technique to recover the full sparsity pattern is to seed the dependency functions with slices of a unit matrix. The run-time τ of the dependency functions is typically orders of magnitude smaller than T. However, for large sparse matrices, the sparsity calculation run-time $\tau\min(n, m)$ could dominate the calculation of the Jacobian. In this work, we propose a hierarchical bitvector-based technique to recover the sparsity pattern faster for highly sparse cases, as would be the case in e.g. multiple-shooting based optimal control problem transcriptions.

*joris.gillis@esat.kuleuven.be; Joris Gillis is a Doctoral Fellow of the Fund for Scientific Research – Flanders (F.W.O.) in Belgium.

The coloring of a sparse Jacobian allows to recover more information from a single sensitivity sweep. A crucial observation is that it can do exactly the same for dependency sweeps. The proposed algorithm starts with obtaining the sparsity pattern in a coarse resolution, performing a coloring of this coarse resolution, and hence potentially reducing the number of fine-grained dependency sweeps needed to obtain a fine-grained image of the sparsity. The algorithm performs this refinement in a recursive way until the full sparsity is recovered:

Input : $\sigma \in \mathbb{N}, \sigma > 1$ subdivision factor
Input : Dimensions n and m of Jacobian
Init : $(N, M) \leftarrow (n, m)$; $r \leftarrow [1]$; `/* Initialize with a scalar */`
while $N > 1$ *and* $M > 1$ **do**
  fwd $\leftarrow$ col(r); adj $\leftarrow$ col(r^T) ; `/* Coloring of the coarse pattern */`
  if *adj is cheaper* **then** seed $\leftarrow$ adj ; $(N, M, n, m, r) \leftarrow (M, N, m, n, r^T)$; mode $\leftarrow$ 'adj';
  else seed $\leftarrow$ fwd; mode $\leftarrow$ 'fwd';
  $(\nu, \mu) \leftarrow$ dimensions of r; $(N, M) \leftarrow (\lceil N/\sigma \rceil, \lceil M/\sigma \rceil)$;
  $S \leftarrow$ block matrix with ν-by-μ empty cells of shape $n/(N\nu)$-by-$m/(M\mu)$;
  foreach $s \in$ *seed* **do**
    $d \leftarrow \text{block\_dep}\left(\text{mode}; s \otimes \mathbf{1}^{m/(M\mu)} \otimes v^M\right)$; `/* Block sparsity seeding */`
    $d \leftarrow \max\left((\mathbf{1}^{n/N} \otimes h^N)d, 1\right)$; `/* Block sparsity aggregate */`
    foreach j *in nonzero locations of* s **do**
      foreach i *in nonzero locations of column* j *of* r **do**
        $S_{i,j} \leftarrow$ rows $ni/(N\nu)$ to $n(i+1)/(N\nu)$ of d; `/* Store result */`
      end
    end
  end
  if *mode* = *'adj'* **then** $S \leftarrow S^T$; $(N, M, n, m) \leftarrow (M, N, m, n)$;
  $r \leftarrow S$;
end
Output: Jacobian sparsity r,

with $\otimes$ the Kronecker product, $\mathbf{1}^n$ a unit matrix of dimension n, v^n a column vector of dimension n with all entries 1, and h^n its transpose. block_dep splits up its bitmatrix argument into columns, feeds these to $\text{dep}_f^{\text{fwd}}$ or $\text{dep}_f^{\text{adj}}$ depending on the mode, and lumps the results back together to form a new bitmatrix. For ease of presentation, the above algorithm is restricted for n and m integer powers of σ. The extension for general dimensions, together with a variant for star-coloring for symmetric Jacobians, was implemented in the `CasADi` framework[1]. In that framework, 64 dependency sweeps are evaluated concurrently and hence a subdivision factor of $\sigma = 64$ was chosen.

The asymptotic run-time is a factor $\sigma/(\sigma - 1)$ worse than the straightforward approach for a fully dense Jacobian (i.e. worst-case). However, for a block-diagonal n-by-n matrix with a blocksize σ, the run-time is $\tau\sigma \log_\sigma(n)$, amounting to a change in complexity from $O(n)$ to $O(log(n))$.
The following table lists run-time results for block-diagonal matrices with blocksize 4-by-4 and shows a clear benefit for the proposed algorithm in practice:

	τ	Run-time, straightforward approach	Run-time, proposed algorithm
$n = 256$	0.11ms	0.6ms (6τ)	0.9ms (8τ)
$n = 16384$	328ms	84.0s (256τ)	1.02s (3τ).

[1] Andersson, J., Åkesson, J., and Diehl, M. CasADi – A symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation* (Berlin, 2012), S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, Eds., Lecture Notes in Computational Science and Engineering, Springer.

[2] Gebremedhin, A. H., Manne, F., and Pothen, A. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review 47* (2005), 629–705.

[3] Giering, R., and Kaminski, T. Automatic sparsity detection implemented as a source-to-source transformation. In *Lecture Notes in Computer Science*, vol. 3994. Springer Berlin Heidelberg, 2006, pp. 591–598.