DSJM: A Software Toolkit for Direct Determination of Sparse Jacobian Matrices[2]

Shahadat Hossain[1]

[1] Department of Mathematics and Computer Science
University of Lethbridge, Canada

5th International Congress on Mathematical Software (ICMS 2016)
(Zuse Institute Berlin (ZIB), Berlin, Germany, July 11 – 14, 2016)

(Joint work with Mahmudul Hasan, Ahmed Imtiaz Khan, Nasrin Hakim Mithila, Ashraful Huq Suny)

[2] Supported in part by NSERC Discovery Grant

Outline

Why Differentiation?

(Measure the sensitivities of design parameters)

Numerical weather forecast was erroneous; determine the initial condition(s) and/or model physical parameter(s) caused discrepancies.
Compute the derivative of the system performance (objective) with respect to the initial condition(s) or parameter(s)

(Optimize a nonlinear function using descent or Newton's method)

Evaluate the gradient and/or the Hessian matrix at points of interest

(Nonlinear least-squares)

Calibration problems in metrology: estimate the calibration parameters given measurement data

(Uncertainty Evaluation)

Evaluate the gradient vector of $f(a)$ to estimate the uncertainty $u(f)$ associated with f from

$$u^2(f) = \nabla f^\top U_a \nabla f$$

Jacobian-vector Products

Given

$$F : \mathbb{R}^n \mapsto \mathbb{R}^m$$

(Finite Difference (FD))

$$\left.\frac{\partial F(x+ts)}{\partial t}\right|_{t=0} = F'(x)s \approx As = \frac{1}{\varepsilon}[F(x+\varepsilon s) - F(x)] \equiv b$$

Forward difference (one extra function evaluation) gives $As = b$ where b is the finite difference approximation.

(Algorithmic Differentiation (AD) (Griewank et al, 2008))

Techniques for evaluating analytic derivatives of computer programs by propagating elementary partial derivatives via chain rule of differential calculus.

1. *bottom-up or forward sweep and top-down or reverse reverse sweep*
2. *implemented as semantic transformation of computer programs: operator overloading and source transformation*
3. *complexity is a small multiple (approx 2–3) of function evaluation*

Algorithmic Differentiation forward mode gives $b = F'(x)s$ accurate up to machine precision

Assumptions

- Given a nonlinear function
$$F : \mathbb{R}^n \mapsto \mathbb{R}^m$$
compute the Jacobian matrix $F'(x)$ at a given x.
- The sparsity pattern of the Jacobian matrix is known a priori and independent of the actual values of x. (Or can be computed using algorithmic differentiation)
- If we need one or more components of F at x we need to compute the whole vector $F(x)$
  - It is more efficient to evaluate the vector $F(x)$ than to evaluate each component of $F(x)$ separately: common sub-expressions are evaluated only once
  - F is a computer subroutine that returns the vector $F(x)$

The CPR Method [Curtis, Powell, and Reid (1974)]

$$A = \begin{pmatrix} 0 & \times & \cdots & 0 & 0 & 0 \\ \times & \times & \cdots & 0 & 0 & 0 \\ \times & 0 & \cdots & 0 & \times & 0 \\ \vdots & \vdots & \cdots & & \vdots & \vdots \\ \times & 0 & \cdots & 0 & \times & \times \\ 0 & \times & \cdots & 0 & 0 & \times \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & j & & & k & \end{matrix}$$

$$F'_j + F'_k \approx A(:, j) + A(:, k) = \frac{1}{\varepsilon}[F(x + \varepsilon(e_j + e_k)) - F(x)]$$

Two columns are **structurally orthogonal** if they do not contain nonzeros in the same row position.

Jacobian Evaluation

Assume ρ_i (number of nonzero unknowns in row i) $\leq p$ for all i

(Compression-Reconstruction procedure for determining A)

.

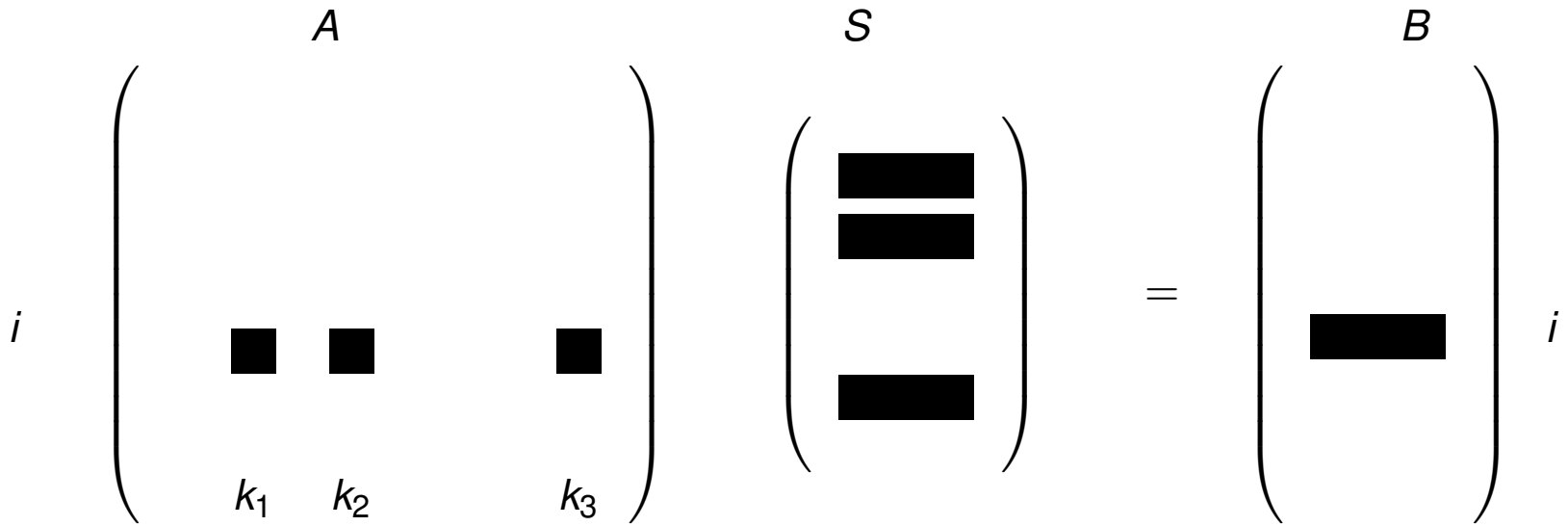
1. ***Seeding or Compression***. *Obtain $S \in \mathbb{R}^{n \times p}$ and evaluate B $(= AS)$ using FD or AD.*
2. ***Harvesting or Reconstruction***. *Determine the nonzero elements of A row-by-row:*
   a. *Identify the reduced seed matrix $\hat{S}_i \in R^{\rho_i \times p}$ for $A(i, \mathcal{J}_i)$ where $\mathcal{J}_i$ denotes a vector containing the column indices of the nonzero entries in row i of A,*

$$\hat{S}_i = S(\mathcal{J}_i, :).$$

   b. *Solve for the ρ_i unknown elements of $A(i, :)$,*

$$\hat{S}_i{}^T A(i, \mathcal{J}_i))^T = B(i, :)^T.$$

Harvesting row i of A



Harvesting the unknown entries with column indices k_1, k_2, and k_3 in row i of A

$$\hat{S}_i^T A(i, \mathcal{J}_i)^T = B(i, :)^T.$$

Direct Determination

For each row of A solve a $p \times \rho_i$ ($p \geq \rho_i$) linear system (usually overdetermined)

S must satisfy the property that each $p \times p$ submatrix of S is invertible.

We say that A **is determined *directly*** *if each $\hat{S}_i$ has a $\rho_i \times \rho_i$ submatrix that is a permuted diagonal matrix.*

Graph Concepts
$A \in \mathbb{R}^{m \times n}$, $G(A) = (V, E)$

$$V = \{A(:,1), \ldots, A(:,n)\}$$

$$E = \{\{A(:,i), A(:,j)| \text{ there is an index } k \text{ for which } A(k,i) \neq 0, A(k,j) \neq 0\}$$

A **p-coloring** of the vertices of G is a function $\phi : V \to \{1, 2, \ldots, p\}$ such that $\{u, v\} \in E \Rightarrow \phi(u) \neq \phi(v)$. The **chromatic number** $\chi(G(A))$ is the smallest p for which $G(A)$ has a p-coloring.

Column partition
A *partition* of the columns of A is a division of columns into groups $C_1, C_2, \cdots, C_p$ such that each column belongs to one and only one group.

Consistent partition

A column partition where each group consists of *structurally orthogonal* columns is called a consistent (with direct determination) partition.

Consistent partitioning and coloring (Coleman and Moré[1983])

- ϕ is a coloring of $G(A)$ if and only if ϕ induces a consistent partition of the columns of A
- Coloring $G(A)$ is as hard as coloring a general graph
- The CPR method is a greedy coloring method
  Consider vertices $v_k = A(:, k)$ in their given order $1, \cdots, n$.

  for $k = 1, \cdots, n$
  Assign vertex v_k the smallest possible color

Observation

Ordering of the vertices affects the coloring – apply greedy coloring to the vertices in some carefully chosen order.

Greedy Partitioning

CPR($\mathcal{S}(A)$, *group*, *ngroup*)

```
1  ngroup ← 1
2  for j ← 1 to n
3       do
4          let L = {l | A(:, j) is not structurally orthogonal to A(:, l)} and
5              c_m = min{c | c ∈ {1, ..., ngroup + 1} ≠ group(l), l ∈ L}
6          group(j) ← c_m
7          if c_m > ngroup
8            then
9                 ngroup ← ngroup + 1
```

Approximating Jacobian-vector Product using FD

FD-SPJD is a user-defined function to compute the difference $F(x+\eta) - F(x)$

FD-SPJMS($gptr, gcolind, k, \eta, B$)

1 $w \leftarrow$ FD-SPJD($F, x, \eta, gptr, gcolind, k$)
2 **for** $ind \leftarrow gptr(k)$ **to** $gptr(k+1) - 1$
3 **do**
4 $j \leftarrow gcolind(ind)$
5 **for** each i for which $F'(i,j) \neq 0$
6 **do**
7 $B(i,k) \leftarrow \frac{w(i)}{\eta(j)}$

Initializing η

1 **for** $ind \leftarrow gptr(k)$ **to** $gptr(k+1) - 1$
2 **do**
3 $j \leftarrow gcolind(ind)$
4 $\eta(j) \leftarrow \epsilon(j)$

Graph Representation and Ordering (Hossain and Steihaug, 2013

Observation

1. $|E| \leq \sum_{1 \leq i \leq m} 0.5\rho_i(\rho_{i-1})$.
2. *$G(A)$ need not be constructed or stored explicitly! Compressed Row and Compressed Column (CRCS) storage stores the cliques:*
   - *Vertices v_j where*

     ```
     k = (rowptr(i): rowptr(i+1)-1);
     j = colind(k);
     ```

     define a clique corresponding to row i of A.
3. *Colpack software (Gebremedhin et al., 2013) uses bipartite graph.*

1. Largest-First Order (LFO)
2. Smallest-Last Order (SLO)
3. Incidence-Degree Order (IDO)
4. Saturation-Degree Order (SDO)
5. Modified Recursive Largest-First Order (MRLF)
6. Hybrid

Main Computational Steps

1. Find the neighbors of column j.
2. Find the next column to be colored (degree calculation).
3. Update the degree information.

1. Use standard compressed row and compressed column data structure for representing the graph
2. Use "bucket" data structure for finding the next node to color. Could use heap (k-ary, Fibonacci).

Storage Format

1. Matrix Market Exchange Format. The row and column indices of the nonzero entries.
2. Harwell-Boeing Exchange Format. FORTRAN based formatted ascii file providing the indices and the values in a compressed form

(The Compressed Row Storage (CRS))

1. **value.** *A double array of size nnz to hold the nonzero entries*
2. **colind.** *An integer array of size nnz to hold the column indices of the nonzero entries*
3. **rowptr.** *An integer array of size* $m + 1$ *that indexes into* **colind** *and* **value**.

Compressed Row Storage (CRS)

$$\begin{pmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & a_{13} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{26} & 0 & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{35} & 0 & 0 & 0 \\ a_{42} & a_{44} & 0 & 0 & 0 & 0 \\ a_{51} & a_{53} & a_{55} & 0 & 0 & 0 \\ a_{62} & a_{64} & a_{66} & 0 & 0 & 0 \end{pmatrix}$$

(a) (b)

Figure: (a) A sparse matrix. (b) Matrix after CRS compression.

Compressed Row Storage (CRS)

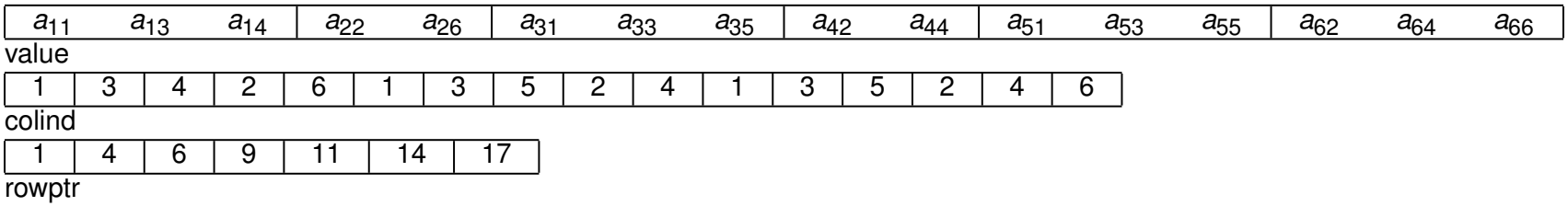


Figure: Compressed Row Storage (CRS) data structure

Accessing row *i*

```
(value(rowptr(i)) : value(rowptr(i+1)-1)).
```

Timing Results ID

(Computing Environment)

Experiments done on IBM PC with 2.8 GHz Intel Pentium CPU, 1 GB RAM, and 512 KB L2 cache running Linux.

Table: Timing Results for IDO Partitioning.

Matrix	*m*	*n*	*nnz*	ColPack		DSJM	
				ot	pt	ot	pt
af23560	23560	23560	484256	1.096	0.324	0.33	0.208
cage11	39082	39082	559722	1.954	0.314	0.472	0.214
cage12	130228	130228	2032536	7.912	1.3	3.018	0.952
lhr71c	70304	70304	1528092	5.324	2.41	1.53	1.618
lpcrea	3516	7248	18168	0.292	0.022	0.038	0.012
lpcreb	9648	77137	260785	14.368	1.498	2.516	0.992
lpcred	8926	73948	246614	14.178	1.524	2.572	1.008
lpdfl001	6071	12230	35632	0.402	0.022	0.05	0.018
lpken11	14694	21349	49058	0.414	0.034	0.072	0.024
lpken13	28632	42659	97246	1.236	0.088	0.188	0.064
lpken18	105127	154699	358171	15.822	0.634	1.5	0.472
lpmarosr7	3136	9408	144848	0.786	0.206	0.186	0.138
lppds10	16558	49932	107605	0.582	0.048	0.12	0.034
lppds20	33874	108175	232647	1.354	0.112	0.258	0.086
lpstocfor3	16675	23541	76473	0.346	0.022	0.04	0.014

Table: Partitioning Results for IDO Partitioning.

Matrix	*m*	*n*	*nnz*	ρ	ColPack	DSJM
af23560	23560	23560	484256	21	41 (SLO)	**37** (MRLF)
cage11	39082	39082	559722	31	62 (SLO)	**54** (MRLF)
cage12	130228	130228	2032536	33	68 (SLO)	**56** (MRLF)
e30r2000	9661	9661	306356	62	68 (LFO)	**65** (MRLF)
e40r0100	17281	17281	553956	62	**66** (LFO)	67 (MRLF)
lpken11	14694	21349	49058	122	123 (IDO)	**122** (MRLF)
lpken13	28632	42659	97246	170	171 (IDO,SLO)	**170** (MRLF, IDO)
lpmarosr7	3136	9408	144848	48	**70** (LFO)	76 (MRLF)

Table: Number of Colors and Required time in seconds for MRLF-SLO , with MRLF running over first $10, 40, 80$ percent of vertices.

Matrix	MRLF	0.1		0.4		0.8	
	Color	Color	Time	Color	Time	Color	Time
af23560	37	41	1.858	40	4.246	37	5.486
cage11	54	62	2.64	60	5.686	59	8.526
cage12	56	67	14.614	65	34.678	63	52.238
e30r2000	65	68	1.988	68	4.88	66	6.042
e40r0100	67	70	3.718	69	9.068	67	11.154
lhr14	63	63	1.09	63	1.844	63	2.168
lhr34	63	63	2.698	63	4.556	63	5.402
lhr71c	63	63	5.376	63	9.11	63	10.814
lpcrea	360	360	0.106	360	0.118	360	0.378
lpcreb	844	845	18.076	845	122.022	844	229.046
lpcred	808	808	20.682	808	135.942	808	246.188
lpdfl001	228	228	0.144	228	0.202	228	0.392
lpken11	122	123	0.186	122	0.208	122	0.45
lpken13	170	171	0.476	170	0.528	170	1.216
lpken18	325	326	3.2	325	3.456	325	12.15
lpmarosr7	76	85	0.62	88	1.092	81	3.608
lpstocfor3	15	15	0.102	15	0.122	16	0.17

DSJM Interface

(C++ Interface)

1. Class `Matrix`: The functionalities of the software are made available to the client through objects of C++ class `Matrix`
2. Input sparsity pattern: Harwell-Boeing and Matrix Market format

```
Matrix matrix(M,N,nnz, false);
int nnz = matrix.compress();
matrix.computeCCS();
matrix.computeCRS();
int *order = new int[N+1];
matrix.slo(order);
int *color = new int[N+1];
int maxgrp = matrix.greedycolor(order,color);
```

(MATLAB Interface)

DSJM *grouping algorithms can be accessed from within MATLAB environment using MATLAB's MEX interface*

```
B = dsjmcolor(A,'slo');
```

Concluding Remarks

(Summary)

- *Implementation is based on array-based data structures*
- *MRLF gives the best coloring nearly always. However, the running time can be much more than the simpler algorithms.*
- *The hybrid MRLF-LFO seems promising.*
- *The code is written in standard C++ ; the coloring routines can be obtained as stand-alone functions.*
- *Successfully interfaced with MATLAB AD Tool MAD (S. Forth).*

(Future Work)

- *Extension to exploit symmetry and two-sided compression are currently being implemented;*
- *Extension to determination by substitution using multiple compression.*

Code can be obtained by contacting the author: shahadat.hossain[at]uleth.ca

Bibliography

T. F. Coleman and J. J. Moré.

Estimation of sparse Jacobian matrices and graph coloring problems.
SIAM J. Numer. Anal., 20(1):187–209, 1983.

A. R. Curtis, M. J. D. Powell, and J. K. Reid.

On the estimation of sparse Jacobian matrices.
J. Inst. Math. Appl., 13:117–119, 1974.

A. H. Gebremedhin, F. Manne, and A. Pothen.

What color is your Jacobian? Graph Coloring for Computing Derivatives.
SIAM Rev., 47(4):629–705, 2005.

Assefaw H. Gebremedhin, Duc Nguyen, Md. Mostofa Ali Patwary, and Alex Pothen.

ColPack: Software for graph coloring and related problems in scientific computing.
ACM Trans. Math. Softw., 40(1):1–31, March 2013.

A. Griewank and A. Walther.

Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.
Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2008.

A. S. Hossain and T. Steihaug.

Computing a sparse Jacobian matrix by rows and columns.
Optimization Methods and Software, 10:33–48, 1998.

Shahadat Hossain and Trond Steihaug.

Graph models and their efficient implementation for sparse jacobian matrix determination.
Discrete Applied Mathematics, 161(12):1747 – 1754, 2013.