A Graph Algorithm for Linearizing Simulink Models

Zhi Han, Pieter J. Mosterman and Fu Zhang

MathWorks, Inc., 3 Apple Hill Dr, Natick, MA 01760, USA

zhi.han|pieter.mosterman|fu.zhang@mathworks.com

Keywords: linearization, simulation, model-based design

Abstract

This paper presents a new efficient approach for performing linearization of Simulink® models. It improves the efficiency of existing linearization algorithms using a Jacobian graph, a graph-based data structure that captures the linear relationship between input, output and state variables. The graph-based algorithm enables the use of graph transformations to reduce the size of the Jacobian data structure, thereby improving the efficiency of subsequent computations. This paper presents a heuristic implementation of the graph-based algorithm. Experimental results on a number of Simulink models of different sizes show how the approach is able to significantly improve computational efficiency and memory use especially in models with large numbers of blocks and states.

1. INTRODUCTION

In recent years, the value of computational models for embedded system design has rapidly increased, for example as the foundation for Model-Based Design (*e.g.*, [1]). As a popular industry tool, Simulink® [2] provides a modeling environment for system design engineers to build complex, linear and nonlinear systems using hierarchical block diagrams. However, most of the analysis and design tools that a control engineer uses deal with linear systems. *Linearization*, that is, obtaining a linear system model for a block diagram model, is often an important step of a model-based analysis and design project. As such, obtaining an accurate linearized model is a crucial step for a successful design.

Simulink provides two sets of algorithms for linearizations: (i) perturbation algorithms and (ii) block-by-block analytic algorithms. With the perturbation algorithms, the linearized model is computed using a finite difference approximation of the Simulink model. With the analytic algorithm, each Simulink block in the model is linearized analytically (provided that such an analytic linearization exists) and stored as an intermediate result; the linearization of the model is obtained from the intermediate results using the chain rule of differentiation. The perturbation algorithm can require more computation time since it involves repeatedly computing the model output for perturbed input values. The block-by-block analytic algorithm is often favored over the perturbation method, and in most cases provides more accurate results. Though it is more efficient in computation time, the block-by-block analytic algorithm typically consumes significantly more memory. For large models, the computer that performs the linearization may run out of physical memory, even if sparse matrices and efficient algorithms for sparse matrices are used. This inefficiency can become severe if linearization is performed many times during a simulation, *e.g.*, in trajectory sensitivity analysis [3].

The current matrix-based block-by-block analytic linearization algorithm can exhibit high memory consumption because it requires that all block inputs and outputs must be first accounted for and must be present in the intermediate results. This paper examines the block-by-block analytic linearization algorithm and proposes a new algorithm based on a newly introduced *Jacobian graph* as an intermediate data structure. The memory requirements of the existing algorithm are alleviated with the Jacobian graph and thus various optimizations can be realized by performing graph transformations on the Jacobian graph. A heuristic approach is applied to a number of Simulink models to illustrate the methodology. The experimental results show significant improvements in terms of memory usage. It is also shown that the additional computation time to perform the reduction step is often small. For large models, due to the reduced size of the matrices in the intermediate results, the computation effort needed for the subsequent steps is also reduced, resulting in less overall computation time.

The Jacobian graph is similar to the computation graph used in numerical analysis [4] and in algorithmic differentiation ([5], [6], [7]). The computational graph plays an important role in the optimization of Jacobian accumulation (*e.g.*, [8], [9]). Graph-based algorithmic differentiation has been successfully applied in software packages [5]. The Jacobian graph introduced in this paper adapts the graph-based algorithmic differentiation methods for use in industrial strength block diagram modeling tools, such as Simulink.

This paper is organized as follows. Section 2. provides background information and discusses the existing block-by-block linearization algorithm implemented in Simulink. Section 3. introduces the concept of a Jacobian graph. Section 4. presents the main results and develops the graph-based algorithm for linearization of a Simulink model. Section 5. applies the algorithm to a number of Simulink models to illustrate the efficiency gains that can be achieved. Section 6. discusses

the limitation of the approach and remaining challenges. Section 7. concludes the paper.

2. ANALYTIC LINEARIZATION IN SIMULINK

To simplify the discussion, this paper considers only the problem of linearizing continuous-time Simulink models. The discussion of linearization of discrete-time models is deferred to Section 6.. For a block implementing a continuous differential equation:

$$\begin{cases} \dot{x} = f(x,u) \\ y = g(x,u) \end{cases} \tag{1}$$

x is called the *state* variable, y is the *output* variable and u is the *input* variable of the block.

Suppose the functions f and g are both differentiable with respect to x and u. The Jacobian of a block is given as the matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} \partial f/\partial x & \partial f/\partial u \\ \partial g/\partial x & \partial g/\partial u \end{bmatrix}. \tag{2}$$

Most Simulink blocks provide a Jacobian function that computes the Jacobian matrix of the block. Computing a linearized model in Simulink using a block-by-block analytic algorithm consists of two steps. First, the Jacobian function of each block is invoked to compute the analytic Jacobians of the blocks (Eqn 2) and store them in memory. Second, the analytic Jacobian of each individual block are aggregated into a number of intermediate results, which are then used to compute the final linearization result of the model. This paper is focused on the second step, known as the *Jacobian accumulation*. Refer to [10] for detailed discussion on the background of block-by-block analytic linearization.

Figure 1 shows the pseudo-code for the model Jacobian accumulation in Simulink. First, Simulink computes a number of intermediate results, including a set of matrices A_o, B_o, C_o, and D_o that aggregate the block Jacobians and a matrix M representing the linear relationship corresponding to the signal connections among blocks. The linearization algorithm then uses the intermediate results to compute the final linearization result using linear-fractional transformation (LFT) [11].

The matrices A_o, B_o, C_o, and D_o consist of matrices of aggregated parts of the block Jacobians. Given an ordered set of blocks $\{blk_1, \cdots, blk_n\}$ with the Jacobian matrices of block blk_i given as $\begin{bmatrix} a_{blk_i} & b_{blk_i} \\ c_{blk_i} & d_{blk_i} \end{bmatrix}$, the matrices A_o, B_o, C_o, and D_o are defined as

Algorithm: `linearize`
Input: A Simulink model S
Output: Model Jacobian A

Algorithm:
Blocks = `get_all_blocks`
Initialize A_o, B_o, C_o, D_o.
`FOREACH` *blk* in *Blocks*:
  Get block Jacobian data
  Split the block Jacobian into matrices a, b, c, and d.
  Add a, b, c and d into A_o, B_o, C_o and D_o.
Initialize M
`FOREACH` *blk* in Blocks:
  Compute $Blocks_{to}$, the set of blocks that *blk* connects to
  `FOREACH` blk_{to} in $Blocks_{to}$:
    Compute $M(blk \mapsto blk_{to})$
    Add $M(blk \mapsto blk_{to})$ to M
$A = \mathcal{F}(A_o, B_o, C_o, D_o, M)$
`RETURN`

Figure 1: Existing algorithm for computing intermediate result

$$A_o = \begin{bmatrix} a_{blk_1} & & \\ & \ddots & \\ & & a_{blk_n} \end{bmatrix} \quad B_o = \begin{bmatrix} b_{blk_1} & & \\ & \ddots & \\ & & b_{blk_n} \end{bmatrix}$$

$$C_o = \begin{bmatrix} c_{blk_1} & & \\ & \ddots & \\ & & c_{blk_n} \end{bmatrix} \quad D_o = \begin{bmatrix} d_{blk_1} & & \\ & \ddots & \\ & & d_{blk_n} \end{bmatrix}.$$

The intermediate results also include a $0/1$-valued matrix M that corresponds to the connection between blocks computed in the following way. A signal connecting the output variable y_{blk_2} and u_{blk_1} establishes the equivalence relations between the variables. For blocks connected directly, the relationship is $u_{blk_1} = y_{blk_2}$. The Jacobian corresponding to the connection is thus $I_{m\times m}$, where m is the length of vectors u_{blk_1} and y_{blk_2}. If there are signal routing blocks involved in the connection, such as the Demux block and Selector blocks illustrated in Fig. 2, the connection must be computed from the logic implemented by the signal routing blocks.

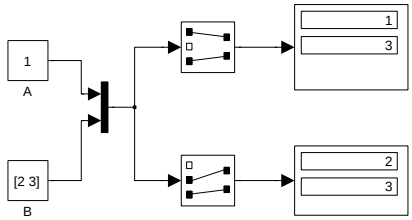


Figure 2: Simulink® supports routing signals using blocks such as Demux and Selector blocks

For two blocks blk_1 and blk_2, consider the connection between output of blk_2 and input of blk_1. Let $y_{blk_2} \in \mathbb{R}^{n_{y_{blk_2}}}$

be the output variable of blk_2 and $u_{blk_1} \in \mathbb{R}^{n_{u_{blk_1}}}$ be the input variable of blk_1. Denote the elements of the variables as y_{i,blk_1} and u_{j,blk_2}. We use $blk_2 \mapsto blk_1$ to denote the set of connected signal elements; and $y_{j,blk_2} \mapsto u_{i,blk_1}$ to denote the element-wise signal connections, if any. The Jacobian of the connection between the output of blk_2 and the input blk_1 is determined by

$$\frac{\partial u_{blk_1}}{\partial y_{blk_2}} = M(blk_2 \mapsto blk_1) \equiv [m_{i,j}]_{n_{u_{blk_1}} \times n_{y_{blk_2}}} \text{ where } m_{i,j} = \begin{cases} 0, & (y_{j,blk_2} \mapsto u_{i,blk_1}) \notin (blk_2 \mapsto blk_1) \\ 1, & (y_{j,blk_2} \mapsto u_{i,blk_1}) \in (blk_2 \mapsto blk_1) \end{cases} \quad (3)$$

If there is no signal connecting the output blk_2 with the input of blk_1, $M(blk_2 \mapsto blk_1) = [0]_{n_{u_{blk_1}} \times n_{y_{blk_2}}}$.

The complete matrix M of the model is thus the aggregated results of the interconnection matrices of all blocks. Note that the connection between the input and output of a Simulink block can be nonzero as Simulink allows self-loops.

$$M = \begin{bmatrix} M(blk_1 \mapsto blk_1) & M(blk_1 \mapsto blk_2) & \cdots & M(blk_1 \mapsto blk_n) \\ M(blk_2 \mapsto blk_1) & M(blk_2 \mapsto blk_2) & \cdots & M(blk_2 \mapsto blk_n) \\ \vdots & \vdots & & \vdots \\ M(blk_n \mapsto blk_1) & M(blk_n \mapsto blk_2) & \cdots & M(blk_n \mapsto blk_n) \end{bmatrix}.$$

Once the intermediate results A_o, B_o, C_o, D_o, M are obtained, the linearization result is computed by performing LFT [11].

$$A = \mathcal{F}(A_o, B_o, C_o, D_o, M) \equiv A_o + B_o M (I - D_o M)^{-1} C_o.$$

Note that we assume the matrix $(I - D_o M)^{-1}$ is invertible. In practice the Simulink software reports an error if the matrix is singular. For correctness of the results, refer to a standard text book on robust control such as [11].

3. JACOBIAN GRAPH

A *Jacobian graph* captures the linear relationship between input, output, and state variables in a graph-like manner.

Definition 1 (Jacobian graph)**.** *A Jacobian graph is defined by the tuple $\mathcal{J} \equiv (\mathcal{V}, \mathcal{E}, Var, \mathcal{K})$ where*

1. *$(\mathcal{V}, \mathcal{E})$ is a directed graph, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges, for $(\mathrm{v}_1, \mathrm{v}_2) \in \mathcal{E}$, we also write $\mathrm{v}_1 \to \mathrm{v}_2$;*
2. *$Var = X \cup Y$ a set of variables, where X is a set of state variables and Y a set of algebraic variables. Each variable is one-to-one correspondent with the vertices $\mathrm{v}_i \in \mathcal{V}$. We also write $Var(\mathrm{v}_i)$ as the variable corresponding to v_i and write $\mathcal{V}(y)$ as the vertex corresponding to variable y.*
3. *$\mathcal{K} : E \to \mathbb{R}^{m \times n}$ a function that assigns a real matrix to each edge. For $\mathcal{K}(v_i, v_j) = K$, we write $v_i \xrightarrow{K} v_j$ as a short-hand notation.*

Note that the edges in the Jacobian graph are denoted by $\to$ as they are distinguished from signal connections of input/output variables of Simulink blocks, denoted by $\mapsto$. The Jacobian graph is a directed graph that may contain loops, including self loops. Repeated edges (*i.e.*, two different edges with the same source and destination vertices) are not allowed in Jacobian graphs.

Fig. 3 shows the Jacobian graph for the continuous-time block given in Eqn (1). The vertices for state variables are shown in a circle shape. And vertices for algebraic variables are shown in rectangles. The state variable x has a self loop which is labeled with the state matrix a. The input variable u and output variable y are algebraic variables. The edges $x \xrightarrow{c} y$, $u \xrightarrow{d} y$ and $u \xrightarrow{b} x$ represent the corresponding submatrices in the block Jacobian.

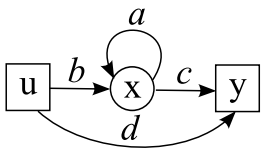


Figure 3: The Jacobian graph of a Simulink® continuous-time block

For block connections $blk_2 \mapsto blk_1$, the Jacobian graph is $\mathrm{v}_{y_{blk_2}} \xrightarrow{M(blk_2 \mapsto blk_1)} \mathrm{v}_{u_{blk_1}}$, where $\mathrm{v}_{y_{blk_2}} = \mathcal{V}(y_{blk_2})$ and $\mathrm{v}_{u_{blk_1}} = \mathcal{V}(u_{blk_1})$.

For a Simulink model consisting of a set of connected blocks, its Jacobian graph is constructed by aggregating the Jacobian graphs for the blocks and the Jacobian graphs for the signal connections. This is accomplished by *summing* the Jacobian graphs of the blocks and signal connections.

Definition 2 (Sum)**.** *For Jacobian graph $\mathcal{J}_1 = (\mathcal{V}_1, \mathcal{E}_1, Var_1, \mathcal{K}_1)$ and $\mathcal{J}_2 = (\mathcal{V}_2, \mathcal{E}_2, Var_2, \mathcal{K}_2)$, the sum of the graphs is defined as $\mathcal{J} = \mathcal{J}_1 + \mathcal{J}_2 = (\mathcal{V}, \mathcal{E}, Var_1 \cup Var_2, \mathcal{K})$, where*

1. *$\mathcal{V} = (\mathcal{V}_1 \cup \mathcal{V}_2)$ and $\mathcal{E} = (\mathcal{E}_1 \cup \mathcal{E}_2)$*
2. *$\mathcal{K}$ is given by the following*

$$\mathcal{K}(i,j) = \begin{cases} \mathcal{K}_1(i,j) & \text{if } (i,j) \in E_1 \wedge (i,j) \notin E_2 \\ \mathcal{K}_2(i,j) & \text{if } (i,j) \notin E_1 \wedge (i,j) \in E_2 \\ \mathcal{K}_1(i,j) + \mathcal{K}_2(i,j) & \text{if } (i,j) \in E_1 \wedge (i,j) \in E_2 \\ \emptyset & \text{otherwise}. \end{cases}$$

For a Jacobian graph $(\mathcal{V}, \mathcal{E}, Var, \mathcal{K})$, an ordered subset of vertices $I_\mathrm{v} \subseteq \mathcal{V}$ and an ordered subset of vertices $O_\mathrm{v} \subseteq \mathcal{V}$, denote the corresponding variables ordered by the vertex ordering as $I = \{i_1, i_2, \cdots, i_m\} \subseteq Var$ and $O = \{o_1, o_2, \cdots, o_n\} \subseteq$

Var, respectively. We define the *submatrix* of the Jacobian graph for I_ν and O_ν as the block matrix

$$sub(\mathcal{J}, I_\nu, O_\nu) = \begin{bmatrix} K_{o_1,i_1} & K_{o_1,i_2} & \cdots & K_{o_1,i_m} \\ K_{o_2,i_1} & K_{o_2,i_2} & \cdots & K_{o_2,i_m} \\ \vdots & \vdots & \cdots & \vdots \\ K_{o_n,i_1} & K_{o_n,i_m} & \cdots & K_{o_n,i_m} \end{bmatrix}$$

where

$$K_{o_j,i_k} = \begin{cases} \mathcal{K}(i_k,o_j) & \text{, if } \mathcal{V}(i_k) \to \mathcal{V}(o_j) \\ \mathbf{0} & \text{, otherwise.} \end{cases}$$

The following theorem establishes the relationship between the Jacobian graph and the intermediate results used in the algorithm `linearize`.

Theorem 1. *For a model consisting of an ordered set of interconnected blocks indexed by* $\{1,...n\}$*, let* $U = \{u_{blk_1},\cdots,u_{blk_n}\}$ *denote the input variables of the blocks,* $Y = \{y_{blk_1},\cdots,y_{blk_n}\}$ *denote the output variables of the blocks, and* $X = \{x_{blk_1},\cdots,x_{blk_n}\}$ *denote the state variables. Let* $\mathcal{J} = (\mathcal{V},\mathcal{E},X\cup(U\cup Y),\mathcal{K})$ *denote the Jacobian graph corresponding to the set of blocks. We use* $\mathcal{X}$*,* $\mathcal{Y}$ *and* $\mathcal{U}$ *to denote the ordered vertices corresponding to X, Y and U, respectively. Then*

$$\begin{aligned} A_o &= sub(\mathcal{J},\mathcal{X},\mathcal{X}) \\ B_o &= sub(\mathcal{J},\mathcal{U},\mathcal{X}) \\ C_o &= sub(\mathcal{J},\mathcal{X},\mathcal{Y}) \\ D_o &= sub(\mathcal{J},\mathcal{U},\mathcal{Y}) \\ M &= sub(\mathcal{J},\mathcal{Y},\mathcal{U}) \end{aligned} \tag{4}$$

Proof. The result follows from the definition of submatrices, the definition of the Jacobian graph for Simulink blocks (Fig 3) and the `linearize` algorithm (Fig. 1). □

4. MAIN RESULT

Figure 4 shows the pseudo-code to compute linearization of Simulink models using Jacobian graph data structure. The algorithm first creates a Jacobian graph $\mathcal{J}$ for the given model using the procedure `generate_jacobian_graph` (Fig. 7). If no optimization is performed, the algorithm uses the intermediate results A_o, B_o, C_o, D_o and M to compute the linearized model. Theorem 1 establishes that algorithm `graph_linearize` produces identical results as the algorithm `linearize` (Fig. 1). If graph transformation is performed, Jacobian graph $\mathcal{J}$ is transformed to produce $\mathcal{J}_r$. Then the submatrices A_r, B_r, C_r and D_r of $\mathcal{J}_r$ are used to compute the linearized model. The algorithm uses *vertex elimination* to reduce the size of the Jacobian graph [5]. The correctness of the algorithm is established by the properties of vertex eliminations [12, 5].

Algorithm: `graph_linearize`
Input: A Simulink model S
Output: Model Jacobian A

Algorithm:
$\mathcal{J}$ = `generate_jacobian_graph`(S)
`IF reduce_graph`:
  $\mathcal{J}_r = \mathcal{J}$
  `WHILE` true:
    ν = `find_next_reducible_vertex`
    `IF` ν is found:
      $\mathcal{J}_r = \mathcal{J}_r \backslash \nu$
    `ELSE`:
      `BREAK`
  Compute the set of state vertices $\mathcal{X}$ and the set of algebraic vertices $\mathcal{Y}_r$ from J
  $A_r = sub(\mathcal{J},\mathcal{X},\mathcal{X})$
  $B_r = sub(\mathcal{J},\mathcal{Y}_r,\mathcal{X})$
  $C_r = sub(\mathcal{J},\mathcal{X},\mathcal{Y}_r)$
  $D_r = sub(\mathcal{J},\mathcal{Y}_r,\mathcal{Y}_r)$
  # *Let* n_Y *denote the sum of widths of variables Y*
  $A = \mathcal{F}(A_r,B_r,C_r,D_r,I_{n_Y\times n_Y})$
`ELSE`:
  Compute the set of state vertices $\mathcal{X}$, the block input vertices $\mathcal{U}$ and the output vertices $\mathcal{Y}$ from J
  $A_o = sub(\mathcal{J},\mathcal{X},\mathcal{X})$
  $B_o = sub(\mathcal{J},\mathcal{Y},\mathcal{X})$
  $C_o = sub(\mathcal{J},\mathcal{X},\mathcal{Y})$
  $D_o = sub(\mathcal{J},\mathcal{Y},\mathcal{Y})$
  $M = sub(\mathcal{J},\mathcal{Y},\mathcal{U})$
  $A = \mathcal{F}(A_o,B_o,C_o,D_o,M)$
`RETURN`

Figure 4: Linearization algorithm using Jacobian graph

In this paper the definition of vertex elimination is slightly modified to accommodate the case where the vertex is involved in an algebraic loop, *i.e.*, a loop involving algebraic variables only.

Definition 3 (Vertex Elimination)**.** *For Jacobian graph* $\mathcal{J} = (\mathcal{V},\mathcal{E},Var,\mathcal{K})$ *and an algebraic variable y with corresponding vertex* ν*, the reduced graph is defined as the reduction* $J_r = J\backslash\nu$ *where* J_r *is the result of the algorithm* `reduce_vertex` *(Fig. 5).*

In the algorithm `reduce_vertex`*, the operation* $\backslash$ *is the standard set difference operation* $A\backslash B = A\cap\bar{B}$*. In the reduced graph* $\mathcal{J}_r$*, the vertex* ν *is removed as well as the edges that connect to* ν*. For every pair of incoming and outgoing edge at* ν*, a new edge is created that bypasses* ν*. The function* $\mathcal{K}'$ *gives the updated* $\mathcal{K}$ *value for the newly created edges. Its value is defined as*

$$\mathcal{K}'(\nu,K_e,K_f) = \mathcal{F}\left(\begin{bmatrix} \mathbf{0} & K_f \\ K_e & D_\nu \end{bmatrix}, I\right)$$

Algorithm: `reduce_vertex`
Input: Jacobian graph $\mathcal{J}$, A vertex ν
Output: Jacobian graph $\mathcal{J}$

Algorithm:
`FOREACH` pair (e, f) such that $e \xrightarrow{K_e} \nu$ and $\nu \xrightarrow{K_f} f$:
  $\mathcal{J} = \mathcal{J} + (e \xrightarrow{\mathcal{K}'(y_i, K_e, K_f)} f)$
  $\mathcal{E} = \mathcal{E} \backslash \{(e \to \nu), (\nu \to f)\}$
Remove the self loop at ν, if any.
$\mathcal{V} = \mathcal{V} \backslash \{\nu\}$
`RETURN`

Figure 5: Algorithm to reduce a vertex

where

$$D_\nu = \begin{cases} \mathbf{0}, & \text{there is no self loop at } \nu \\ K_\nu, & \mathcal{K}(\nu, \nu) = K_\nu \end{cases}$$

Figure 6 illustrates the effects of eliminating a vertex ν.

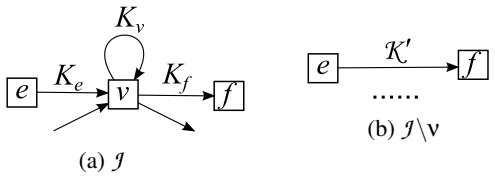


Figure 6: Vertex elimination

The problem of finding the optimal graph transformations for Jacobian accumulation is NP-complete [13, 12]. In this paper a simple heuristic is adopted. Since the purpose of reducing the graph is to reduce the size of the Jacobian graph so as to reduce the size of the intermediate result, one must consider when to perform vertex elimination. In general eliminating a vertex does not guarantee a smaller size Jacobian graph. Suppose a vertex has m incoming edges and n outgoing edges and no self-loop, eliminating this vertex requires eliminating $m+n$ edges and adding mn bypass edges. This paper employs a heuristic for optimizing algebraic variables based on the Markowitz degree ([5]), which is defined as the multiple of the number of incoming and outgoing edges of a vertex in a digraph, excluding self-loops. A variable is eliminated if all of the following conditions are true.

1. Its corresponding variable is an algebraic variable.
2. Its Markowitz degree is less than the sum of the number of incoming and outgoing edges.

If the Markowitz degree of the vertex is less than the number of incoming and outgoing edges, it is provable that vertex elimination will decrease the number of edges. Thus the heuristic for graph transformation always results in a smaller Jacobian graph.

The pseudo-code to construct the model Jacobian graph is shown in Fig. 7. The construction of the Jacobian graph is implemented as a recursive algorithm that recurse on subsystems. Compared to the algorithm `linearize`, which loops over all the blocks in a model, the recursive algorithm `generate_jac_graph` loops through the same set of block in a different order. For the given subsystem, the algorithm loops through the blocks in the subsystem. If a block is a subsystem, the procedure recursively calls itself to compute the Jacobian graph J_s of the subsystem. If a block is a primitive block, the procedure generates the atomic Jacobian subgraph for the block. Then it creates the edges that correspond to the input and output signal connections of that block.

Algorithm: `generate_jac_graph`
Input: A Simulink subsystem S
Output: Jacobian graph $\mathcal{J}$

Algorithm:
B = `get_blocks_in_system`
`FOREACH` block *blk* in B:
  IF *blk* is a subsystem:
    Get S_{blk}, the subsystem corresponding to *blk*.
    $\mathcal{J}_s =$ `generate_jac_graph`(S_{blk})
    $\mathcal{J} = \mathcal{J} + \mathcal{J}_s$
  ELSE:
    $\mathcal{J}_{blk} =$ `create_jacobian_graph`(*blk*)
    $\mathcal{J} = \mathcal{J} + \mathcal{J}_{blk}$
`RETURN`

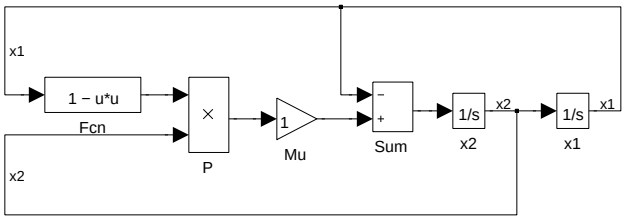
Figure 7: Algorithm for generating Jacobian graphs for subsystems

The use of Jacobian graph and vertex elimination heuristics are illustrated in the following example using a simple Simulink model.

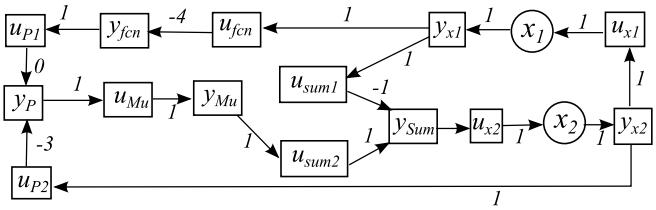
Example 1. *Figure 8 shows a Simulink model (Figure 8a), the Jacobian graphs computed for the model (Fig. 8b) and the reduced Jacobian graph $\mathcal{J}_r$ (Fig 8c). The original Jacobian graph $\mathcal{J}$ has* 16 *vertices and* 18 *edges, whereas the reduced Jacobian graph $\mathcal{J}_r$ has* 2 *vertices and* 3 *edges. All the vertices corresponding to algebraic variables of the models are eliminated using the heuristics.*

5. EXPERIMENTAL RESULTS

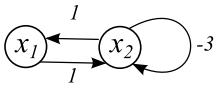
The graph-based algorithm to compute linearization is implemented in the Simulink software. The vertex-reduction heuristics is explored in a prototype implementation. It is used to linearize a number of Simulink models. The Simulink models used in this study span a wide range of scale and complexity. The tested models include small demo models with fewer than 10 blocks to relatively large-scale models with more than 10,000 blocks. The largest model used in this experiment is a Simulink model of the NASA HL-20 spacecraft



(a) The Simulink® model



(b) The Jacobian graph



(c) After vertex reduction

Figure 8: Comparison of Jacobian graph $\mathcal{J}$ and reduced Jacobian graph $\mathcal{J}_r$

automatic landing system [2]. We have also included two industry models to evaluate the impact of this work on sample industry applications. These models are available at MATLAB Central [14]. The experiments are all performed on a computer with Intel Xeon® X5650 CPU at 2.67GHz and 24 GB memory running Debian Linux 6.

Table 1 and Table 2 show the experimental results. The graph size is measured in terms of numbers of edges and numbers of vertices, *e.g.*, `V:12 E:15` indicates that the Jacobian graph has 12 vertices and 15 edges.The experiment shows the reduction in terms of the size of the Jacobian graph, *i.e.*, the total number of vertices and edges. We also compare the reduction of the memory usage of storing the intermediate results of the Jacobian computation algorithm. The intermediate results consist of the matrices A_o, B_o, C_o, D_o and M as well as other auxiliary data. The computation time measures the total computation time of linearization: including computing block Jacobians, constructing the Jacobian graph, reducing the Jacobian graph if graph transformation is performed, computing the intermediate results, and computing the linearization results using LFT.

In both small models and large models, the heuristic implemented in the algorithm results in a significant reduction of the size of the Jacobian graph. Reducing the size of the Jacobian graph leads to a reduction in the memory used for storing the intermediate results. The reduction of memory usage in the structure of intermediate results is less significant than the size reduction of the Jacobian graph. There are several reasons for this. First, the matrices used in the computation are stored as sparse matrices, where the memory used to store the Jacobian matrices depends not only on the number of submatrices and sizes of the submatrices, but also on the sparseness of the submatrices. And the sparseness of the matrices is not completely determined by their size. Second, the Jacobian structure also stores some auxiliary data for keeping track of the states and blocks being linearized. These data are not altered by the variable reduction heuristic and thus remains the same after the reduction. As a result, memory consumption in large-scale models is reduced by a larger percentage.

For small models listed in Table 1, the computation time of the Jacobians is often not measurable because the computation is very fast, and the overhead of performing the vertex elimination is often not detectable. For large models, the additional computation time involved in the reduction step is sometimes greater than the reduction of computation time involved in the later step, resulting in a slight increase of total computation time. For example, the experiment with `asbSkyHogg` model results in a 2.44% increase in the overall computation time. For relatively large-scale models, the vertex elimination saves a considerable amount of memory, and the computation time saved in the subsequent computation often outweighs the additional computation time required for the vertex elimination procedure. The combined effect is a reduced total computation time and reduced memory use. For example, the graph-based computation for the `asbhl20` example model from Aerospace Blockset results in a 92.88% reduction in memory use and 34.36% reduction in total computation time for linearization.

This paper does not measure the complete memory consumption of MATLAB and Simulink to avoid including additional factors into the comparison, such as the memory used by the MATLAB desktop and the memory used by Simulink to render the block diagram. For the models used in the experiment, the measured memory use is much smaller than the complete memory footprint of MATLAB, but the memory used for linearization increases as the model complexity increases.

We also deliberately measured the computation time and memory only for the Jacobian accumulation step. Our observation is that Jacobian accumulation is frequently the most time-consuming for large-scale models [3]. There are other computation steps involved in linearizing a model, such as initializing the model [15]. This computational overhead is always necessary to perform block-by-block linearizations. However, if linearization is to be performed more than once, there are other ways to avoid some of the overhead. See [15] for more details.

Model	# of blocks	Without vertex elimination		With vertex elimination		Memory usage reduction
		$\mathcal{J}$ size	Memory used(bytes)	$\mathcal{J}_r$ size	Memory used (bytes)	
bounce	8	V:12 E:15	1944	V:2 E:4	1832	-5.76%
vdp	13	V:21 E:26	1928	V:4 E:6	1672	-13.27%
foucault	13	V:28 E:34	2758	V:4 E:10	2406	-12.76%
watertank	18	V:27 E:31	2176	V:2 E:4	1792	-17.64%
scdpwm	30	V:30 E:30	2312	V:2 E:2	1944	-15.92%
f14	75	V:103 E:124	6386	V:16 E:36	5074	-20.54%

Table 1: Computation results for Simulink demo models

Model	# of blocks	Without vertex elimination			With vertex elimination			Effects of vertex elimination	
		$\mathcal{J}$ size	Memory used (bytes)	Time (secs)	$\mathcal{J}_r$ size	Memory used (bytes)	Time (secs)	Memory usage reduction	Computation time reduction
aeroblk_wf_3dof	890	V:897 E:1061	258642	0.15	V:26 E:50	6600	0.14	-74.53%	-6.67%
Industry model 1	1070	V:468 E:555	63208	0.1	V:44E:90	53272	0.1	-15.71%	0%
Industry model 2	2059	V:1479 E:1857	525780	0.46	V:119 E:287	30758	0.44	-94.15%	-4.35%
asbSkyHogg	2864	V:2581 E:3209	93990	0.82	V:114 E:181	28950	0.84	-69.20%	+2.44%
power_fivecells	4098	V:1651 E:1961	3128452	0.61	V:125 E:218	2226228	0.56	-28.84%	-8.2%
power_wind_ig	7878	V:7356 E:843	2607036	7.04	V:464 E:596	137480	6.91	-94.73%	-1.85%
power_LF network_29bus	8689	V:6815 E:8530	2996498	11.5	V:808 E:1748	484826	7.08	-83.82%	-38.43%
asbhl20	10622	V:11403 E:15275	3543094	18.13	V:555 E:1298	252332	11.0	-92.88%	-34.36%

Table 2: Computation results for large models

6. DISCUSSION

The existing block-by-block analytic linearization algorithm `linearize` was implemented using sparse matrices [10]. This paper presents a method that improves the efficiency of computation based on the Jacobian graph data structure. The graph-based algorithm is implemented in the Simulink software [2]. A number of implementation details are not discussed in this paper. These include, for example, the handling of linearization input/output point, the handling for datastore memory blocks, and the handling of referenced models. Refer to [2] for technical details on these topics. The vertex-reduction heuristics is explored in a prototype implementation.

This paper limits the vertex elimination algorithm to continuous-time systems. The method is trivially extensible to discrete-time models with single sampling rates. The method is not directly applicable to models with multiple sampling rates. For models with multiple discrete sampling rates, Simulink Control Design™ employs a sample-rate conversion approach. When multiple sampling rates exist in a model, a target sampling rate is given for the linearization algorithm. Blocks with a sampling rate different from the target rate are converted to an equivalent system with the target sampling rate. The algorithm converts blocks in every sampling rate repeatedly until it has converted all the sampling rates into the target sampling rate [10]. Since the sampling rate of the blocks has changed, the resulting model is not semantically identical to the original. Thus this method is not adopted in this paper. Another reason for not adopting the method is that it does not support some Simulink capabilities such as sample time offsets.

This paper focuses on the Jacobian accumulation problem and to simplify the necessary data structure, it is limited in its richness to address this purpose only. The data structure is not intended to capture simulation semantics of the Simulink model or other block diagram modeling languages and includes no specific information to this end. There are a number of publications that explore this direction (*e.g.*, [16], [17], [18], and [19]). The relationship between the semantic graph and Jacobian graph remains to be explored.

7. SUMMARY

This paper introduces the notion of a Jacobian graph, together with a graph-based algorithm to compute the linearized system model for Simulink models. The graph-based approach for model Jacobian accumulation enables one to op-

timize the algorithm for memory usage and computation time without affecting the outcome. Experiments show that both memory use and computation time can be improved using a heuristic approach. The graph-based algorithm unlocks opportunities for future work on developing more efficient algorithms for linearization of Simulink models.

ACKNOWLEDGMENTS

The authors thank Dr. Murali Yeddanapudi and Dr. John Glass for their help and support on this work.

REFERENCES

[1] Gabriela Nicolescu and Pieter J. Mosterman, editors. *Model-Based Design for Embedded Systems*. Model-Based Design for Embedded Systems. CRC Press, Boca Raton, FL, 2009.

[2] MathWorks. *Using Simulink®*. The MathWorks, Inc., Natick, MA, September 2012.

[3] Zhi Han and Pieter J Mosterman. Towards sensitivity analysis of hybrid systems using Simulink. In *Hybrid Systems: Computation and Control (HSCC)*, pages 95–100, 2013.

[4] Friedrich Ludwig Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

[5] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principle and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[6] H. M. Bücker and C. H. Bischof. What computers can do for you automatically: The role of automatic differentiation in inverse modeling. *GACM Report*, Spring 2011(6):25–28, 2011.

[7] Sri Hari Krishna Narayanan, Boyana Norris, Paul Hovland, Duc C. Nguyen, and Assefaw H. Gebremedhin. Sparse Jacobian computation using ADIC2 and ColPack. *Procedia Computer Science*, 4:2115 – 2123, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[8] Andreas Griewank. A mathematical view of automatic differentiation. In *Acta Numerica*, volume 12, pages 321–398. Cambridge University Press, 2003.

[9] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, Philadelphia, PA, 1991.

[10] MathWorks. *Simulink Control Design: Block-by-Block Analytic Linearization*. MathWorks, Inc., Natick, MA, September 2012.

[11] Kemin Zhou and John C. Doyle. *Essentials of Robust Control*. Prentice Hall, 1998.

[12] Uwe Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming*, 99:399–421, 2004.

[13] Uwe Naumann. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming*, 112(2):427 – 441, 2008.

[14] MathWorks. MATLAB central file exchange. http://www.mathworks.com/matlabcentral/fileexchange.

[15] MathWorks. *Simulink Control Design: Speeding up linearization of complex models*. MathWorks, Inc., Natick, MA, September 2012.

[16] Ben Denckla, Pieter J. Mosterman, and Hans Vangheluwe. Towards an executable denotational semantics for causal block diagrams. In *In OOPSLA 05 Workshop on Domain-Specific Modeling*, 2005.

[17] Pieter J. Mosterman, Justyna Zander, Gregoire Hamon, and Ben Denckla. Towards computational hybrid system semantics for time-based block diagrams. In *IFAC Converence on analysis and design of hybrid systems*, 2009.

[18] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21 – 42, 2003.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.