Jacobian Pattern Synthesis and Application for Dynamic System Ensembles Using Boolean Linear Fraction Transformation

Fu Zhang, Zhi Han, and Pieter J. Mosterman
MathWorks, 3 Apple Hill Dr, Natick, MA 01760, USA
`fu.zhang|zhi.han|pieter.mosterman@mathworks.com`

Keywords: Model-based design, Jacobian pattern, Dependency graph, Boolean LFT

Abstract

Simulation and analysis of complex, large-scale dynamic systems can be challenging. Complex system structures are difficult for systems engineers to explore and comprehend, and simulating large-scale systems often requires long computation time because of the size of the system equations. This paper introduces the *Jacobian pattern* for continuous dynamic systems. A *Boolean linear fractional transformation* (BLFT) computation is developed to compute the Jacobian pattern for a block diagram model by extending the existing Jacobian accumulation algorithm in the modeling and simulation environment Simulink. Applications illustrate faster simulation as well as the ability to better analyze and comprehend how variables on a dynamic system model affect one another.

1. INTRODUCTION

Modern control emerged in the 1960s based on a state-space formulation of dynamic systems. This formulation captures the output of a dynamic system as a function of input, as well as internal state. Moreover, the evolution of the internal state is explicitly captured as a derivative with respect to time. This time derivative is a function of input and state as well. In a linear form this leads to the well-known system of equations

$$\begin{array}{rcl} \dot{x} & = & Ax + Bu \\ y & = & Cx + Du \end{array} \tag{1}$$

where the output, input, and state are vectors $y \in \mathbb{R}^m$, $u \in \mathbb{R}^k$, $x \in \mathbb{R}^n$, respectively, with $\dot{x}$ denoting the time derivative of the state. As this mathematical formulation of control systems became increasingly well understood, ever more powerful analysis, design, and synthesis of controllers were facilitated.

Models of the dynamic behavior of physical systems often require in nonlinear functions. This leads to the more general representation as a system of ordinary differential equations (ODEs):

$$\dot{x} = f(t, x, u) \tag{2a}$$

$$y = g(t, x, u) \tag{2b}$$

To apply mathematical techniques developed for the linear representation in Eq. (1), the nonlinear form must be linearized, which can be based on the Jacobian. Suppose that both f and g are differentiable with respect to x and u; the Jacobian matrix of Eq. (2) is defined as:

$$J = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \equiv \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial u} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial u} \end{bmatrix} \tag{3}$$

This Jacobian provides the linearized elements necessary to arrive at the linear form of Eq. (2) [1]. Similarly, the Jacobian of a dynamic system is important in system analysis. It can, for example, enable stiffness analysis [2, 3] and sensitivity analysis [4].

While relatively straightforward for a given system of nonlinear equations in state-space form, control systems nowadays consist of many such systems that have their input and output connected to comprise extensive models, almost invariably including hierarchical structures. Tools for Model-Based Design (e.g., [5, 23]) such as Simulink® [6] help engineers solve problems in the design of complex control, signal processing, and communications systems. Engineers construct models of complex dynamic systems as interconnections of fundamental blocks, such as gain, integrator, transformation, and so on. An important role is assumed by *subsystem* blocks, which themselves contain blocks to define the subsystem behavior, thus providing a hierarchical decomposition language element [18].

The convenience with which such block diagrams can be created has led to the design of models with more than a million blocks. As a result, an engineer may request analyses between model variables that involve a large number of connected systems of equations. If the analysis is numerical in nature and relies on a linear formulation, this requires an overall Jacobian of the connected blocks between selected model variables.

Complementary to numerical analyses, the complexity of block diagram models renders a structural dependency analysis of great importance as well. First, it provides engineers with insight into which variables are affected in their behavior by other variables. This structural dependency information is useful to accelerate model execution by exploring the structure of the dynamic equations, as discussed in Section 5.

In other applications the dependencies are exploited in the generation of imperative code. For example, modeled functionality may be partitioned in a subset of classes that minimize the potential for dependency cycles [20, 22]. Such cycles may be resolved by heterogeneous function composition based on the dependency analysis as well [21].

Moreover, dependency analysis may serve as the foundation of sophisticated reasoning methods, such as those employed in model-based diagnosis [19]. Compared to a numerical Jacobian, if a structural dependency is required, a corresponding Boolean-valued pattern Jacobian (also referred to as *Jacobian pattern*) may suffice, where in first order the pattern may establish whether there is or there is not a dependency. More sophisticated qualitative patterns may identify positive or negative temporal dependencies [19].

This work addresses the challenge of deriving the dependencies between input/output relationships of arbitrary points in a complex block diagram model. This is based on synthesizing a pattern Jacobian in a closed-form; that is, the open loop pattern Jacobian of all pertinent blocks combined with their input/output connections is reduced to the overall dynamic system's pattern Jacobian.

Synthesizing the pattern Jacobian of a block diagram based model involves three critical challenges. (i) The closed-form of the overall model's system equation and the relation among variables are embedded in the block diagram. (ii) Model can be very complicated, and contain many level of hierarchy. They may even contain other models (for example, by model reference blocks). (iii) Models may contain user-written blocks that provide inaccurate Jacobian or pattern Jacobian information.

This paper presents a block-by-block method to synthesize the pattern Jacobian of a Simulink model, and the applications of the pattern in simulating and analyzing the models. The remaining sections of the paper are arranged as follows. In Section 2., background of the pattern Jacobian and Simulink semantics are discussed. In Section 3., the block-by-block method to compute the Jacobian is reviewed and the well-known *Linear Fraction Transformation* (LFT) method is introduced. Section 4. proposes the Boolean LFT that computes the pattern Jacobian. In Section 5., applications of the pattern Jacobian are discussed. Section 6. summarizes the contribution of the paper.

2. BACKGROUND

Generally, the Jacobian matrices of a dynamic system are time-varying and state-dependent. However, some of the numerical entries of Jacobian matrices are always 0, for example, because of the structure of the system. These entries are categorized as *hard* 0s. This allows one to form a time invariant pattern of the Jacobian matrix that captures the structural dependencies between the system variables (input, output, state, and the time derivative of the state).

For a system equation given as in Eq. (2) and Jacobian defined as in Eq. (3), a Boolean-valued pattern Jacobian matrix, also referred to as a *Jacobian pattern* matrix, is defined as:

$$J_p = \begin{bmatrix} A_p & B_p \\ C_p & D_p \end{bmatrix}, \text{where } J_{p(i,j)} = \begin{cases} 0, \text{if } J_{(i,j)}(t) = 0 \\ 1, \text{otherwise} \end{cases} \forall\, t. \tag{4}$$

Matrix J_p can be computed from J in Eq. (3) by converting all non-hard 0 entries in the matrix J to 1 and maintaining all hard 0 entries as 0. The resulting 0 entries describe the sparsity of J. Often many of the overall entries in J_p are 0 and J_p is a sparse matrix.

For example, consider the following equation:

$$\begin{cases} \dot{x}_1 = (x_1)^2 + 2x_2 + u_1 \\ \dot{x}_2 = x_2 + u_2 \\ \dot{x}_3 = x_3 \end{cases} \\ \begin{cases} y_1 = 2x_1 + (x_2)^2 + (u_2)^2 \\ y_2 = x_3 \end{cases} \tag{5}$$

The Jacobian matrix can be computed as

$$J = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} 2x_1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 2 & 2x_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 2u_2 \\ 0 & 0 \end{pmatrix} \end{bmatrix}, \tag{6}$$

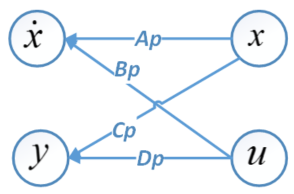
and the Jacobian pattern matrix is

$$J_p = \begin{bmatrix} A_p & B_p \\ C_p & D_p \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \end{bmatrix} \tag{7}$$

Using A_p as an example, the blocks of the Jacobian pattern matrix can be studied. Let $a_{p(i,j)}$ be an entry of A_p, then $a_{p(i,j)} \equiv 0 \rightarrow \frac{\partial f_i}{\partial x_j} = \frac{\partial \dot{x}_i}{\partial x_j} \equiv 0$. This relationship means that $\dot{x}_i$ cannot be influenced by x_j, indicating there is no dependency between $\dot{x}_i$ and x_j. Whereas the A_p matrix captures the dependency between $\dot{x}_i$ and x, B_p does so for the dependency between $\dot{x}_i$ and u, C_p for the dependency between y and x, and D_p for the dependency between y and u.

Overall, J_p shows the relation between the left-hand side variables $(\dot{x}_i, y)$ and the right-hand side variables (x, u). In other words, J_p describes the causality between variables (x, u) and variables $(\dot{x}_i, y)$. For example, if $a_{p(i,j)} = 1$, then x_j is needed to compute $\dot{x}_i$.

From the Jacobian pattern matrix, a dependency graph can be constructed as a directed graph $G = (V, E)$ in which $\dot{x}_i, x, y, u$ form nodes V, and the directed edges from x and u to $\dot{x}_i$ and u are represented by the corresponding pattern matrices A_p, B_p, C_p, and D_p. The dependency graph G of Eq. (2) and its adjacency matrix [1] is shown in Figure 1.



$$\begin{array}{c|cccc} & \dot{x} & x & y & u \\ \dot{x} & 0 & A_p & 0 & B_p \\ x & 0 & 0 & 0 & 0 \\ y & 0 & C_p & 0 & D_p \\ u & 0 & 0 & 0 & 0 \end{array}$$

Figure 1. The dependency graph and adjacency matrix of Eq. (2)

Simulink enables computing J and J_p from a physical system model that is implemented in software based on a dynamic system in the state-space formulation. Typically, a fundamental Simulink block implements a system of differential equations as Eq. (2). That is, a block implements (i) an output method that corresponds to Eq. (2a) to compute block output $y(t)$ from $x(t)$ and $u(t)$, and (ii) a derivative method that corresponds to Eq. (2b) to compute $\dot{x}(t)$ from $x(t)$ and $u(t)$. Moreover, a block may implement a Jacobian method to compute the Jacobian of Eq. (3) and Jacobian pattern of Eq. (4) [6, 7].

For complex models, the block diagram is usually hierarchical. That is, a subsystem block that represents part of the model is built first and several such subsystem blocks are connected to form the block diagram of the overall model. Each of the subsystem blocks may comprise fundamental blocks as well as further subsystem blocks.

The fundamental and subsystem blocks are connected by directed lines from a block output to a block input. The lines represent the block input and output variables.

Executing or simulating a block diagram computes a time trajectory of the variables that are interesting to the user. A numerical solver is used to integrate the state and push time forward. Simulation involves executing the output and derivative method of a block and integrating the derivatives to solve the system states. One typical operational semantics of simulating a block diagram is to define the system output method as aggregation (with a given sorted order) of every block's output, and the system derivative method as an aggregation of every block's derivative method and form a simulation loop with the numerical solver. A detailed discussion of such semantics can be found in [7]. Such execution semantics may be called a block-by-block method because the overall model's method is "synthesized" from each of the blocks' methods.

[1]Normally, the adjacency matrix is defined as $[m_{i,j}] = 1$ if there is an edge from vertex i to vertex j in E. The definition in this paper switches the index i and j, so Jacobian pattern matrices are adjacency matrices.

3. LFT AND BLOCK-BY-BLOCK METHOD TO COMPUTE THE JACOBIAN

This section describes an LFT-based block-by-block algorithm to compute the Jacobian of the dynamic system as Eq. (2) modeled with Simulink. Details of this algorithm are also presented in other work [1, 8].

3.1. Linear Fractional Transformations

There are two types of linear fractional transformation, the lower LFT and the upper LFT [16]. This paper uses the lower LFT (the procedure is analogous when using an upper LFT).

Definition 1 (lower *Linear Fractional Transformation*) [9] For a block matrix

$$J = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \in \mathbb{R}^{(m_1+m_2)\times(n_1+n_2)}$$

and a matrix $E \in \mathbb{R}^{n_2\times m_2}$ the (lower) LFT of J with respect to E is defined as

$$\mathcal{F}_l(J,E) \equiv A + BE(I-DE)^{-1}C \tag{8}$$

where $I \in \mathbb{R}^{m_2\times m_2}$ is an identity matrix.

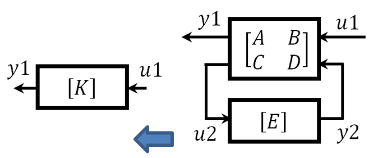


Figure 2. Linear fractional transformation

In other work [16], the LFT has been used to simplify a closed-loop system described by a block diagram. Figure 2 shows how the LFT can be used to simplify a block diagram that contains two blocks representing the following equations:

$$b1 : \begin{cases} y_1 = Au_1 + Bu_2 \\ y_2 = Cu_1 + Du_2 \end{cases}$$

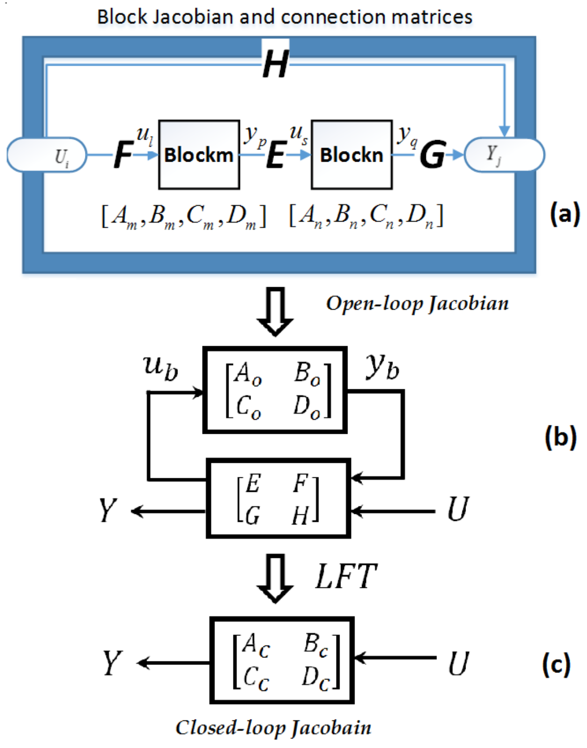
and

$$b2 : y_2 = Eu_2$$

where A, B, C, D, and E are matrices while y and u are output and input vectors, respectively. On the right in Fig. 2 is a closed-loop block diagram formed with the two blocks, with connections between them. Using LFT, the ensemble of blocks can be simplified to a gain block with $K = A + BE(I-DE)^{-1}C$. Both of these two block diagrams represent the same relation between output y_1 and input u_1.

3.2. A Block-by-Block Method

The block-by-block algorithm to compute the Jacobian consists of three steps and is depicted in Figure 3.



Block Jacobian and connection matrices
H
U_i
F
u_l
Blockm
y_p
E
u_s
Blockn
y_q
G
Y_j
$[A_m, B_m, C_m, D_m]$
$[A_n, B_n, C_n, D_n]$
(a)
Open-loop Jacobian
u_b
y_b
$\begin{bmatrix} A_o & B_o \\ C_o & D_o \end{bmatrix}$
(b)
$\begin{bmatrix} E & F \\ G & H \end{bmatrix}$
Y
U
LFT
$\begin{bmatrix} A_C & B_c \\ C_C & D_C \end{bmatrix}$
Y
U
(c)
Closed-loop Jacobain

Figure 3. Connection matrices of a Simulink model

Step 1: Construct the open-loop Jacobian and the connection matrices In this step, each block is linearized first by computing its Jacobian. In Simulink, this is done by invoking each block's Jacobian method. Jacobian matrices of each block $A_i, B_i, C_i, D_i (i = 1, 2, ...n)$ are then concatenated into a set of block diagonal matrices. These matrices, which contain every block's Jacobian, are called the open-loop Jacobian

$$J_O = \begin{bmatrix} A_O & B_O \\ C_O & D_O \end{bmatrix} \tag{9}$$

where

$$A_O = \begin{bmatrix} A_1 & & \\ & \ddots & \\ & & A_n \end{bmatrix} \quad B_O = \begin{bmatrix} B_1 & & \\ & \ddots & \\ & & B_n \end{bmatrix}$$
$$C_O = \begin{bmatrix} C_1 & & \\ & \ddots & \\ & & C_n \end{bmatrix} \quad D_O = \begin{bmatrix} D_1 & & \\ & \ddots & \\ & & D_n \end{bmatrix}$$

Simulink also computes a set of connections matrices that represent the linear relationships corresponding to the block input/output connections and model-level input/output connections. Model-level input/output connections are the blocks that represent the input and output of the Simulink block diagram as a model component itself. The set of all possible connections can be categorized as connections:

- from block output y to block input u (E),
- from model-level input U to block input u (F),
- from block output y to model-level output Y (G), and
- from model-level input U to model level output Y (H).

Notice that the connection matrices E, F, G, and H are all Boolean-valued matrices. Construction of these matrices is shown in Fig. 3(a).

Step 2: Form the internal linear representation After each block is linearized, inside the model is a set of linear blocks that are connected together. The system equations of this linear model can be written as:

$$\begin{aligned} \dot{x} &= A_O x + B_O u_b \\ y_b &= C_O x + D_O u_b \end{aligned} \tag{10}$$

where u_b is a vector of all block input variables, y_b is a vector of all block output variables, and x is a vector of all the states in the model. Eq. (10) shows the linear relations between block input u_b and block output y_b.

Also, the interconnection relations among all block input/output and model input/output variables can be written

$$\begin{aligned} u_b &= E y_b + F U \\ Y &= G y_b + H U \end{aligned} \tag{11}$$

where U and Y are vectors of model input and output variables, respectively. Here Eq. (10) and Eq. (11) form a diagram with a closed-loop structure, which is depicted in Fig. 3(b).

Step 3: Compute the closed-loop Jacobian J_C using the LFT In this step, the closed-loop diagram formed in step 2 is transformed applying the LFT to the block diagram that represents the linear form of the model, with the state vector x and model input U and model output Y, as depicted in Figure 3(c).

The result of the transform is a model Jacobian that is also called the closed-loop Jacobian of the model, J_C, where

$$J_C = \begin{bmatrix} A_C & B_C \\ C_C & D_C \end{bmatrix} = \begin{bmatrix} \frac{\partial \dot{X}}{\partial X} & \frac{\partial \dot{X}}{\partial U} \\ \frac{\partial Y}{\partial X} & \frac{\partial Y}{\partial U} \end{bmatrix} \tag{12}$$

The model Jacobian J_C describes the sensitivity among the model level variables $(\dot{X}, X, Y, U)$. By using the LFT algorithm, the submatrices of the closed-loop model Jacobian matrix J_C are computed as

$$\begin{aligned} A_C &= A_O + B_O E (I - D_O E)^{-1} C_O \\ B_C &= B_O (F + E (I - D_O E)^{-1} D_O F) \\ C_C &= G (I - D_O E)^{-1} C_O \\ D_C &= H + G (I - D_O E)^{-1} D_O F \end{aligned} \tag{13}$$

Note that the internal model variables (i.e., the blocks' input and output variables u_b and y_b, respectively) are eliminated by the LFT algorithm.

4. BOOLEAN LFT AND JACOBIAN PATTERN

Unlike the numerical Jacobians, the Jacobian pattern matrices are Boolean matrices. This section shows that a similar algorithm, called *Boolean LFT*, can be used with Boolean matrices to compute the model Jacobian pattern based on the block Jacobian pattern.

4.1. Boolean LFT Algorithm

The following definitions are necessary in the proceedings.

Definition 1 (Boolean Arithmetic): If a and b are binary digits (0 or 1), then

$$\begin{aligned} a \wedge b &= \begin{cases} 1, \text{if } a = b = 1 \\ 0, \text{otherwise} \end{cases} \\ a \vee b &= \begin{cases} 0, \text{if } a = b = 0 \\ 1, \text{otherwise} \end{cases} \end{aligned} \tag{14}$$

Definition 2 (Boolean matrix): Matrix M is a Boolean matrix if $M = [m_{i,j}]$, $m_{i,j} \in \{0,1\}$.

Definition 3 (Boolean matrix addition operator $\oplus$):

$$M \oplus N = [m_{i,j} \vee n_{i,j}] \tag{15}$$

Definition 4 (Boolean matrix multiplication operator $\otimes$): Let two Boolean matrices $M = [m_{i,j}]$ be $p \times q$, and $N = [n_{i,j}]$ be $q \times l$, the Boolean matrix multiplication is defined by

$$M \otimes N = [\bigvee_{k=1}^{n} m_{i,k} \wedge n_{k,j}] \tag{16}$$

Definition 5 (Power of Boolean matrix): The power n exponentiation of square Boolean matrix M is

$$M^n = \underbrace{M \otimes M \otimes \cdots \otimes M}_{n} \tag{17}$$

It is straightforward to prove that a Boolean matrix, M, has these properties:

$$M \oplus M = M \tag{18}$$

Other work contains Boolean matrix details [10, 11].

Definition 6 (Transitive closure of Boolean Matrix): The edges of a directed graph $G = (V,E)$, $n = |V|$, specify paths of length 1 between pairs of vertices. This graph can be represented by the Boolean $n \times n$ *adjacency matrix* $M = [m_{i,j}]$, $1 \le i,j \le n$, where $m_{i,j} = 1$ if there is an edge from vertex j to vertex i in E, otherwise $m_{i,j} = 0$.

A Boolean matrix M^* is called the *transitive closure* of M, if M^* whose i,j entry $m^*_{i,j} = 1$, if there is a path of length ≥ 0 from vertices j to i in G, and $m^*_{i,j} = 0$ otherwise. Algorithms to find the transitive closure Boolean Matrix can be found in other work [11, 12].

Definition 7 (Pattern operator): A pattern operator P returns the pattern of a matrix $M \in \mathbb{R}^{p \times q}$.

$$P(M) = M_p = \begin{cases} 0, \text{if } m_{i,j} \equiv 0 \\ 1, \text{otherwise} \end{cases} \tag{19}$$

M_p is the pattern of M.

If zeros caused by coincidence numerical cancellation (*soft* 0s) are prohibited, then the pattern operator P has the following properties:

$$\begin{aligned} P(A+B) &= A_p \oplus B_P \\ P(A \times B) &= A_P \otimes B_P \\ P(A^n) &= (A_p)^n \end{aligned} \tag{20}$$

Also, it can be shown that ([13], Corollary 5.4):

$$P(A^{-1}) = (P(A))^* = (A_p)^* \tag{21}$$

which means the pattern of the inverse of A is the transitive closure of A_p.

Now if the P operator is applied to Eq. (13), the LFT algorithm results in the model closed-loop Jacobian. Let $A_{Cp} = P(A_C), A_{Op} = P(A_O), B_{Op} = P(B_O), C_{Op} = P(C_O)$,and $D_{Op} = P(D_O)$, then $P(A_C) = P(A_O + B_O E (I - D_O E)^{-1} C_O))$ or

$$P(A_C) = P(A_O) \oplus [P(B_O) \otimes E \otimes P((I - D_O E)^{-1}) \otimes P(C_O)]$$

Notice that $P((I - D_O E)^{-1}) = ((P(I - D_o E))^* = (I \oplus D_{Op} \otimes E)^*$. Then

$$\begin{aligned} A_{Cp} &= A_{Op} \oplus [B_{Op} \otimes E \otimes (I \oplus D_{Op} \otimes E)^* \otimes C_{Op}] \\ B_{Cp} &= B_{Op} \otimes [F \oplus (E \otimes (I \oplus D_{Op} \otimes E)^* \otimes C_{Op})] \\ C_{Cp} &= G \otimes (I \oplus D_{Op} \otimes E)^* \otimes C_{Op} \\ D_{Cp} &= H \oplus [G \otimes (I \oplus D_{Op} \otimes E)^* \otimes C_{Op}] \end{aligned} \tag{22}$$

Eq. (22) is called the *Boolean LFT* algorithm, which is a block-by-block method to compute the model Jacobian pattern. The Boolean LFT is similar to the LFT expression in Eq. (13), which, by overloading the multiply and addition operator, enables code reuse when programming the code for computing both Jacobian J and Jacobian pattern J_p in an object-oriented language such as C++.

Finally, the LFT definition is extended to the Boolean domain.

Definition BLFT (Lower *Boolean Linear Fractional Transformation*): Let $\mathbb{B} = \{0,1\}$, for a Boolean block matrix

$$J = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \in \mathbb{B}^{(m_1+m_2)\times(n_1+n_2)}$$

and a matrix $E \in \mathbb{B}^{n_2 \times m_2}$, the (lower) Boolean LFT of J with respect to E is defined as

$$\mathcal{F}_{lb}(J,E) \equiv A \oplus [B \otimes E \otimes (I \oplus D \otimes E)^* \otimes C] \tag{23}$$

where $I \in \mathbb{B}^{m_2 \times m_2}$ is a Boolean identity matrix.

4.2. Boolean LFT and Dependency Paths

The Jacobian pattern describes the dependency (or reachability) between variables. It will be shown that the *Boolean LFT* can find all possible paths from one variable to another by exploring the internal structure (connections) of a model.

Using the expression to find A_{Cp} as an example, the expression $A_{Cp} = A_{Op} \oplus [B_{Op} \otimes E \otimes (I \oplus D_{Op} \otimes E)^* \otimes C_{Op}]$ contains an obvious structure that shows all the possible paths along which a derivative variable $\dot{x}_i$ can be reached from a state variable x_j. These paths involve: (i) A_{Op}, (ii) B_{Op} and C_{Op}, and (iii) B_{Op}, D_{Op}, C_O, and E.

By replacing each block in a model with its dependency graph and then connecting the dependency graphs, the model dependency graph is formed, as shown in Fig. 4.

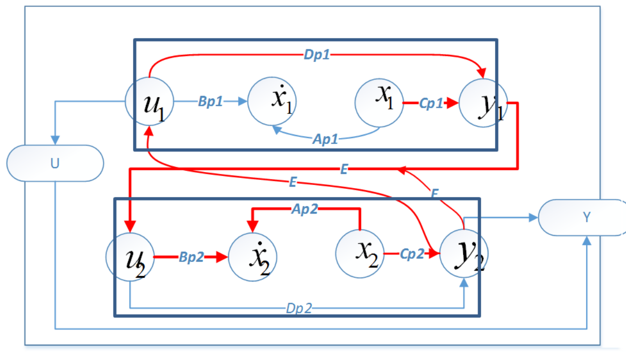


Figure 4. Boolean LFT and its graphical meaning

From the model graph it can be found that there are three paths by which a derivative variable $\dot{x}_i$ can be reached from (or depends on) a state variable x_j.

1. **Internal path within a block**:

$$\dot{x}_i \xleftarrow{A_{Op}} x_j$$

   This path maps to A_{Op} in the Boolean LFT expression of A_{Cp}. The path length is 1.

2. **Short external path**:

$$\dot{x}_i \xleftarrow{B_{Op}} u \xleftarrow{E} y \xleftarrow{C_{Op}} x_j$$

   For this path, x_i and x_j could be in the same or different blocks. This path maps to $B_{Op} \otimes E \otimes C_{Op}$ in the Boolean LFT expression of A_{Cp}. The path length is 3.

3. **Long external path**:

$$\dot{x}_i \xleftarrow{B_{Op}} u \xleftarrow{E} y \xleftarrow{D_{Op}} u \xleftarrow{E} y \xleftarrow{C_{Op}} x_j$$

   This path maps to $B_{Op} \otimes E \otimes (D_{Op} \otimes E)^* \otimes C_{Op}$ in the Boolean LFT expression of A_{Cp}. The path length is ≥ 5.

As a result the Boolean LFT can find all possible paths through which $\dot{x}_i$ can be reached from x_j. This information about all paths could not be explored from A_{Cp} because an entry '1' in A_{Cp} indicates that there is at least one path from x_j to $\dot{x}_i$, but not which path specifically or how many. That said, if all that is required is a Boolean dependency relation, then as long as a $a_{Cp(i,j)} = 1$ any remaining paths are irrelevant.

Once J_{Cp} is computed it can be used for many simulation applications.

5. JACOBIAN PATTERN APPLICATIONS

The Jacobian pattern analysis can be applied to achieve a broad range of objectives, such as more efficient simulation and structural analysis.

5.1. Efficient Jacobian Computation

If Eq. (2) is to be solved by an implicit ODE solver, then the ODE solver must compute $(I - hJ_x)^{-1}$, where J_x equals matrix A of Eq. (3) and h is the integration step size.

When analytic block Jacobians are available, the solver computes the model Jacobian using *LFT* as discussed in Section 3. When the blocks do not provide analytic Jacobians, a perturbation method must be used to compute an approximated Jacobian. However, compared to analytical block Jacobians, the Jacobian patterns are much easier to obtain. As such, the Jacobian patterns are usually available, even when the analytic block Jacobians are not.

If the pattern of J_x is not known, the entries of the q^{th} column of J_x can be approximated as:

$$j_{i,q} = \left[\frac{\partial f_i}{\partial x_q}\right] = \frac{f_i(x_p, x_q + \Delta x_q) - f_i(x_p, x_q)}{\Delta x_q} \qquad (24)$$

where $i = 1, 2, \ldots, n$ and $p = 1, 2, \ldots, n, p \neq q$. This method is called the *full perturbation* method. For this method, to compute J_x, the state vector must be perturbed n times and the derivative equation must also be evaluated n times.

The Simulink solver implements an efficient Jacobian perturbation algorithm by precomputing the model Jacobian pattern. When the Jacobian pattern of J_x is computed, the solver computes the approximated Jacobian using the following steps:

First, partition the columns of the pattern matrix into groups, if the p^{th} column $j_{i,p}$ and q^{th} columns $j_{i,q}$ of the pattern matrix satisfy $j_{i,p} \circ j_{i,q} = 0$, where $i = 1, 2, \ldots, n$ and $\circ$ is the vector dot product of vectors. Second, the states belong to the same column group can be perturbed together to compute J_x.

This two-step algorithm is called the *sparse perturbation* method. If the number of groups m is less than n, then the time to compute J_x may be decreased. When J_x is a very sparse matrix the sparse perturbation could be much faster than full perturbation. It is worth mentioning that there are many ways to find the column groups of a sparse matrix, the method to acquire the minimum number of column groups is equivalent to K-coloring a graph [14].

For example, the Jacobian pattern J_{xp} of Eq. (5) is $\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. It can be found that one grouping scheme is to group column 2 and 3 together, because $\begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T = 0$. J_x can be computed using sparse perturbation as

$$\begin{aligned} &\text{perturb} \quad x_1 \left\{ \left[\frac{\partial f_{1,2,3}}{\partial x_1}\right] = \frac{f_{1,2,3}(x_1+\Delta x_1, x_2, x_3)}{\Delta x_1} \right. \\ &\text{perturb } x_2, x_3 \left\{ \begin{aligned} \left[\frac{\partial f_{1,2,3}}{\partial x_2}\right] &= \frac{f_{1,2,3}(x_1, x_2+\Delta x_2, x_3)}{\Delta x_2} \\ \left[\frac{\partial f_{1,2,3}}{\partial x_3}\right] &= \frac{f_{1,2,3}(x_1, x_2, x_3+\Delta x_3)}{\Delta x_3} \end{aligned} \right. \end{aligned}$$

where x_2, x_3 are perturbed simultaneously in one function evaluation. The model derivative method only needs to be evaluated twice. Another group scheme is to group column 1 and 3 together, which results in the same number of function evaluations.

After J_x is computed, the implicit solver must solve $(I - hJ_x)^{-1}$. Usually this step is done using LU factorization. If the Jacobian pattern is not known, the computational complexity of this operation is $O(n^3)$. With the Jacobian pattern information it is possible to reduce the complexity, but the exact value is a complicated function of the pattern. Other work presents details about sparse factorization performance along with numerical results [15, 17].

Figure 5 shows the sparse pattern of a Simulink model of a hydraulic system. There are 327 states and only 6 column groups. Where simulation of this model using full perturbation takes 56 seconds, with sparse perturbation it takes only 44 seconds. The experiments are performed on a computer with an Intel Xeon® processor at 2.67 GHz and 4 GB memory running MATLAB R2014a.

5.2. Structural analysis

A Simulink model contains abundant system information that provides insight into the system dynamics. For example, a mechanical engineer working on a train design may want to know if the velocity of the train is affected by other state variables. In hierarchical block diagram models, tracing these dependencies can be difficult as (a) there may be many blocks involved in the data path; (b) the presence of blocks that group and ungroup signals may cause confusion in visual inspection: the signal connection indicates that there are dependencies, but numerical computation has no dependency at all. The Jacobian pattern solves this problem as it is computed for the closed-loop model, where the many blocks in the data path are reduced, and the dependency is computed for every

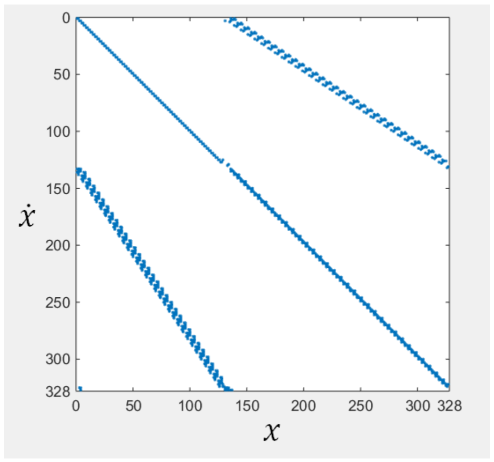


Figure 5. Solver Jacobian pattern of a hydraulic system with 327 states but only 6 columm groups

state variable. For example, Figure 6 shows the block diagram of the *f14* model in Simulink and its corresponding solver Jacobian pattern [6]. Finding dependencies among variables directly from the block diagram is not easy because of levels of hierarchy introduced by subsystems. However, from the first row of the solver Jacobian pattern matrix a user can easily find that the derivative of vertical velocity (x_1) could be affected by x_1 itself as well as the pitch rate (x_2), the actuator output (x_3), and the wind gust (x_4 and x_5). This information can provide insight when studying the system or verifying of the model.

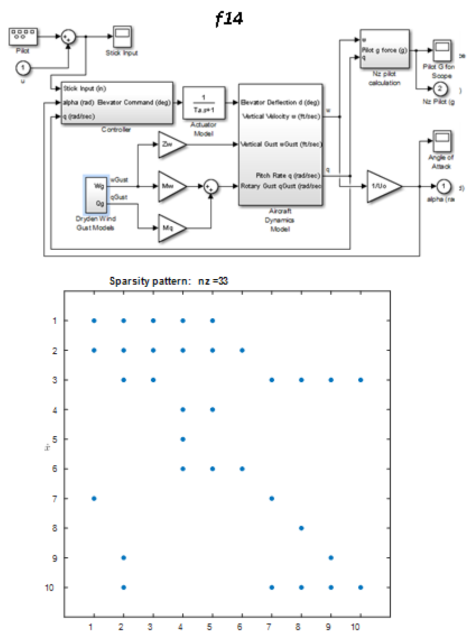


Figure 6. f14 and its solver Jacobian pattern

6. SUMMARY

This paper presents the concept of a *Jacobian pattern* of a dynamic system and its connections to the dependency graph or structure of the system, followed by the introduction of a *Boolean LFT* and a block-by-block algorithm to compute the Jacobian patterns of Simulink models. Applications of the Jacobian pattern are then discussed, which include accelerating simulation, exploring the dependency of the system, and verifying the correctness of the model.

REFERENCES

[1] Zhi Han, Pieter J. Mosterman, and Fu Zhang."A graph algorithm for linearizing Simulink models." *Proceedings of the 2013 Summer Computer Simulation Conference*, 2013.

[2] Curtis, A. R. "Jacobian matrix properties and their impact on choice of software for stiff ODE systems." IMA journal of numerical analysis 3.4 (1983): 397-415.

[3] Higham, Desmond J., and Lloyd N. Trefethen. "Stiffness of ODEs." BIT Numerical Mathematics 33.2 (1993): 285-303.

[4] 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.

[5] 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.

[6] MathWorks. Using Simulink. The MathWorks, Inc., Natick, MA, September 2012.

[7] Bouissou, Olivier, and Alexandre Chapoutot. "An operational semantics for Simulink's simulation engine." ACM SIGPLAN Notices 47.5 (2012): 129-138.

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

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

[10] Kim, Ki Hang. Boolean matrix theory and applications. Vol. 70. New York: Dekker, 1982.

[11] Fischer, Michael J., and Albert R. Meyer. "Boolean matrix multiplication and transitive closure." Switching and Automata Theory, 1971., 12th Annual Symposium on. IEEE, 1971.

[12] Savage, John E. Models of computation. Vol. 136. Reading, MA: Addison-Wesley, 1998.

[13] Gilbert, John R. "Predicting structure in sparse matrix computations." SIAM Journal on Matrix Analysis and Applications 15.1 (1994): 62-79.

[14] Hossain, AKM Shahadat, and Trond Steihaug. "Computing a sparse Jacobian matrix by rows and columns." Optimization Methods and Software 10.1 (1998): 33-48.

[15] Davis, Timothy A., and Iain S. Duff. "An unsymmetric-pattern multifrontal method for sparse LU factorization." SIAM Journal on Matrix Analysis and Applications 18.1 (1997): 140-158.

[16] Houlis, P. & Sreeram, V., "An Interconnection between Combined Classical Block Diagrams and Linear Fractional Transformation Block Diagrams.", in 'ICARCV', IEEE (2006), pp. 1-5 .

[17] Gupta, Anshul. "Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices." SIAM Journal on Matrix Analysis and Applications 24.2 (2002): 529-552.

[18] Péter Fehér, Tamás Mészáros, László Lengyel, and Pieter J. Mosterman. "A Novel Algorithm for Flattening Virtual Subsystems in Simulink Models." ICSSE 2013, pp. 369-375, Budapest, Hungary, July 4-6, 2013.

[19] Pieter J. Mosterman, Ravi Kapdia, and Gautam Biswas. "Using bond graphs for diagnosis of dynamic physical systems." DX-95, pp. 81-85, October 2-4, 1995.

[20] Marc Pouzet and Pascal Raymond. "Modular Static Scheduling of Synchronous Data-flow Networks: An Efficient Symbolic Representation." EMSOFT'09 pp. 215-224, Grenoble, France, 2009.

[21] Ben Denckla and Pieter J. Mosterman. "An intermediate representation and its application to the analysis of block diagram execution." *Proceedings of the 2004 Summer Computer Simulation Conference* (SCSC'04), pp. 167-172, July 25-29, 2004.

[22] Roberto Lublinerman and Stavros Tripakis. "Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams." DATE'08, pp. 1504-1509, March 10-14, Munich, Germany, 2008.

[23] Pieter J. Mosterman and Justyna Zander. "Advancing Model-Based Design by Modeling Approximations of Computational Semantics," in *Proc. of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pp. 3-7, September 5, 2011