**List of changes**


Replaced: statements . . . . . 4
Replaced: statement . . . . . 4
Added: Change k index for j in **a** so . . . . . 5
Deleted: as this . . . . . 7
Replaced: partial derivatives . . . . . 9
Replaced: vertex . . . . . 12
Replaced: by extending features of . . . . . 16

# Compact Sparse Symbolic Jacobian Computation in Large Systems of ODEs.

Ernesto Kofman[a,b], Joaquín Fernández[b*], Denise Marzorati[a*]

[a] *FCEIA-UNR, Argentina*
[b] *CIFASIS-CONICET, Argentina*

**Abstract**

This work introduces a novel algorithm that automatically produces computer code for the calculation of sparse symbolical Jacobian matrices. More precisely, given the code for computing a function $\mathbf{f}$ depending on a set of state (independent) variables $\mathbf{x}$, where the code makes use of intermediate algebraic (auxiliary) variables $\mathbf{a}(\mathbf{x})$, the algorithm automatically produces the code for the symbolic computation of the matrix $J = \partial\mathbf{f}/\partial\mathbf{x}$ in sparse representation.

A remarkable feature of the algorithm developed is that it can deal with iterative definitions of the functions preserving the iterative representation during the whole process up to the final Jacobian computation code. That way, in presence of arrays of functions and variables, the computational cost of the code generation and the length of the generated code does not depend on the size of those arrays. This feature is achieved making use of *Set–Based Graph* representation.

The main application of the algorithm is the simulation of large scale dynamical systems with implicit Ordinary Differential Equation (ODE) solvers like CVODE-BDF, whose performance are greatly improved when they are invoked using a sparse Jacobian matrix. However, the algorithm can be used in a more general context for solving large systems of nonlinear equations.

The paper, besides introducing the algorithm, discusses some aspects of its implementation in a general purpose ODE solver front-end and analyzes some results obtained.

*Keywords:* Large Scale Models, Jacobian Computation, Set–Based Graphs

## 1. Introduction

Large scale systems of ODEs are usually stiff so they need implicit ODE solvers to be efficiently simulated [1]. These numerical integration algorithms use the Jacobian matrix to solve the nonlinear implicit equations involved and their performance is largely improved when the Jacobian matrix is externally provided with a sparse form representation [2, 3].

In a simple PDE discretization, the code for the sparse Jacobian computation can be manually obtained without much effort. However, in more complex models where there are several interacting subsystems, the manual task becomes almost impossible. Motivated by this problem (and similar problems in

*Corresponding author
*Email address:* kofman@cifasis-conicet.gov.ar (Denise Marzorati[a])

a more general context) several algorithms in the field of *algorithmic differentiation* have been developed [4, 5, 6].

A usual approach for computing the Jacobian (either in dense or sparse form) is to build a graph representing the dependences between the function components $f_i$ and the state and auxiliary variables. Then, the different paths from a function component $f_i$ to a given state $x_j$ represent different terms that accumulate in the expression of $\partial f_i/\partial x_j$. These terms can be computed using the chain rule as the path traverses different algebraic variables.

To the best of our knowledge, all the existing algorithms assume that function $\mathbf{f}$ is composed by individual scalar functions $f_i$ depending on individual states $x_j$ and algebraic variables $a_k$. Thus, in large scale systems the generation of the code for computing a Jacobian matrix becomes computationally expensive as it depends at least linearly in the number of variables involved. In models containing thousands or millions of states, the computational cost associated to the code generation and the subsequent compilation of the resulting huge piece of code eventually make the problem unsolvable. So the use of a symbolic Jacobian is only possible if it can be manually obtained.

However, large scale systems are (almost) always defined making use of repeating structures (typically 'for' statements) and arrays of variables. That way, there are sets of functions with identical definitions where the only thing that changes between components are the indexes of the arrays they involve. Exploiting this feature is the main purpose of the present work.

For that goal, we propose first to represent the dependence between functions and variables making use of *Set–Based Graphs* (SBG) [7], where each vertex (called set–vertex) can represent an entire array of variables or functions and each edge (called set–edge) represents all the connections between pairs of set–edges. This allows a compact representation which results independent on the size of the arrays involved. Then, based on this compact representation, we develop an algorithm that produces the code to compute the sparse Jacobian matrix. A remarkable property is that the computational cost of this algorithm and the length of the code it produces is independent on the size of the arrays.

The algorithm was implemented as part of a tool called *Stand Alone QSS Solver* [8] that has a front end to different ODE solvers, including CVODE, DASSL, DOPRI, as well as the whole family of Quantized State System (QSS) methods. This tool allows defining the models using a subset of the Modelica language [9] and translates them into plain C code providing also the code for the symbolic computation of the dense or sparse Jacobian (for algorithms like DASSL and CVODE) and structural information (incidence matrices) for QSS algorithms. The tool performs all the transformations preserving the for–loop statements so that the code produced does not depend on the size of the arrays involved in the model.

The article is organized as follows. After the formal statement of the problem below, Section 2 introduces some previous results and tools that are used in the remaining of the work. Then, Section 3 presents the novel algorithm developed and Section 4 discusses its implementation. Finally, the performance of this implementation is studied in Section 5 and some conclusions and future research ideas are proposed in Section 6.

*1.1. Problem Formulation*

We consider a model written in the following explicit form:

$$\begin{aligned} a_1 &= g_1(x_1, \ldots, x_n, t) \\ a_2 &= g_2(x_1, \ldots, x_n, a_1, t) \\ &\vdots \quad \vdots \\ a_m &= g_m(x_1, \ldots, x_n, a_1, \ldots, a_{m-1}, t) \\ \dot{x}_1 &= f_1(x_1, \ldots, x_n, a_1, \ldots, a_m, t) \\ &\vdots \\ \dot{x}_n &= f_n(x_n, \ldots, x_n, a_1, \ldots, a_m, t) \end{aligned} \tag{1}$$

which defines an ODE

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

Our goal is to automatically produce the computer code that calculates the Jacobian matrix

$$\mathcal{J}(x) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \tag{3}$$

expressed in sparse matrix representation.

When a system like that of Eq.(1) is large, it usually contains repeating structures making use of arrays of unknowns and equations (defined using `for loop` statements~~commands~~). In that case, the algebraic and state variables are grouped in different arrays $\mathbf{a^1}, \ldots \mathbf{a^m}$, $\mathbf{x^1}$, $\mathbf{x^n}$. Similarly, the functions defining the algebraic variables and state derivatives can be grouped in arrays such that all the individual functions that share the same definition inside a `for loop` statement~~command~~ belong to the same array.

In consequence, the model can be rewritten as

$$
\begin{aligned}
a^1_{i+s_g^{1,1}} &= g_i^{1,1}(\mathbf{x^1},\ldots,\mathbf{x^n},t), \quad i=1,\ldots \\
a^1_{i+s_g^{1,2}} &= g_i^{1,2}(\mathbf{x^1},\ldots,\mathbf{x^n},a_1^1,\ldots,a_k^1,t), \quad i=1,\ldots; \quad k<i+s_g^{1,2} \\
&\vdots \quad \vdots \\
a^m_{i+s_g^{m,1}} &= g_i^{m,1}(\mathbf{x^1},\ldots,\mathbf{x^n},\mathbf{a^1},\ldots,\mathbf{a^{m-1}},t), \quad i=1,\ldots \\
&\vdots \quad \vdots \\
a^m_{i+s_g^{m,j}} &= g_i^{m,j}(\mathbf{x^1},\ldots,\mathbf{x^n},\mathbf{a^1},\ldots,\mathbf{a^{m-1}},a_1^m,\ldots,a_k^m,t), \quad i=1,\ldots; k<i+s_g^{m,j} \\
&\vdots \quad \vdots \\
\dot{x}^1_{i+s_f^{1,1}} &= f_i^{1,1}(\mathbf{x^1},\ldots,\mathbf{x^n},\mathbf{a^1},\ldots,\mathbf{a^m},t), \quad i=1,\ldots \\
\dot{x}^1_{i+s_f^{1,2}} &= f_i^{1,2}(\mathbf{x^1},\ldots,\mathbf{x^n},\mathbf{a^1},\ldots,\mathbf{a^m},t), \quad i=1,\ldots \\
&\vdots \quad \vdots \\
\dot{x}^n_{i+s_f^{n,j}} &= f_i^{n,j}(\mathbf{x^1},\ldots,\mathbf{x^n},\mathbf{a^1},\ldots,\mathbf{a^m},t), \quad i=1,\ldots \\
&\vdots \quad \vdots
\end{aligned}
\tag{4}
$$

Change k index for j in **a** so that the explanation and the $x$ indexes are the same

Here $s_g^{m,j}$ is an index shift (with $s_g^{m,1}=0$)) so that the $i+s_g^{m,j}$ component of $\mathbf{a}^m$ is defined by the $i$–th component of $\mathbf{g^{m,j}}$. An analogous role is played by $s_f^{m,j}$for computing the state derivatives.

The purpose of this work is to design and to implement an algorithm that produces the code for computing the Jacobian matrix of Eq.(4) such that

- The computational cost of producing the code does not depend on the size of the different arrays involved.
- The length of the code produced does not depend on the size of those arrays.
- The code generated is efficient in the sense that it does not repeat unnecessary calculations, but it is not necessary optimal in any other sense.

In addition, the code produced should be able to compute the Jacobian in sparse representation.

## 2. Background

In this section we present some previous results and tools that are used along the rest of the paper.

*2.1. Automatic Jacobian Matrix Computation*

The problem of producing the code for computing the Jacobian matrix of a function lies in the discipline of *automatic differentiation* (AD), or, more precisely, in the field of *algorithmic differentiation* [5].

The literature on AD mainly focus of producing efficient code for computing the derivatives of functions, including here the problem of the sparse Jacobian computation [10, 4, 11]. The goal of the discipline is to produce an efficient code to compute the Jacobian (and other higher derivatives), for which the community has developed several techniques and approaches. However, to the best of our knowledge, the problem of producing a compact piece of code for very large systems has not been tackled so far.

In this work, we shall only make use of some existing AD techniques in order to obtain the code for computing scalar partial derivatives which will be part of the final Jacobian computation. Yet, the approach is independent on the way those derivatives are obtained.

*2.2. Set–Based Graphs*

The algorithms presented in this work are based on the use of *Set-Based Graphs* (SB-Graphs), first defined in [7]. SB-Graphs are regular graphs in which the vertices and edges are grouped in sets allowing sometimes a compact representation. We introduced next the main definitions.

**Definition 1** (Set–Vertex)**.** *A* Set–Vertex *is a set of vertices* $V = \{v_1, v_2, \ldots, v_n\}$.

**Definition 2** (Set–Edge)**.** *Given two Set–Vertices,* $V^a$ *and* $V^b$*, with* $V^a \cap V^b = \emptyset$*, a Set–Edge connecting* $V^a$ *and* $V^b$ *is a set of non repeated edges* $E[\{V^a, V^b\}] = \{e_1, e_2, \ldots, e_n\}$ *where each edge is a set of two vertices* $e_i = \{v_k^a \in V^a, v_l^b \in V^b\}$.

**Definition 3** (Set–Based Graph)**.** *A Set–Based Graph is a pair* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *where*

- $\mathcal{V} = \{V^1, \ldots, V^n\}$ *is a set of disjoint set–vertices (i.e.,* $i \neq j \implies V^i \cap V^j = \emptyset$*).*
- $\mathcal{E} = \{E^1, \ldots, E^m\}$ *is a set of set–edges connecting set–vertices of* $\mathcal{V}$*, i.e.,* $E^i = E[\{V^a, V^b\}]$ *with* $V_a \in \mathcal{V}$ *and* $V_b \in \mathcal{V}$*. In addition, given two set edges* $E^i, E^j \in \mathcal{E}$ *with* $i \neq j$*, such that* $E^i = E[\{V^a, V^b\}]$ *and* $E^j = E[\{V^c, V^d\}]$*, then* $V^a \cup V^b \cup V^c \cup V^d \neq V^a \cup V^b$*. This is, two different set–edges in* $\mathcal{E}$ *cannot connect the same set–vertices.*

A particular case of Set–Based Graph is a bipartite Set–Based Graph defined as follows:

**Definition 4** (Bipartite Set–Based Graph)**.** *A Bipartite Set–Based Graph is a Set–Based Graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *where two disjoints sets of set–vertices* $\mathcal{V}_1$*,* $\mathcal{V}_2$ *can be found verifying* $\mathcal{V}_1 \cup \mathcal{V}_2 = \mathcal{V}$*, such that for every edge* $E^i = E[\{V^a, V^b\}] \in \mathcal{E}$ *the condition* $V^a \in \mathcal{V}_i$ *implies that* $V^b \notin \mathcal{V}_i$

As in regular graphs, Set–Based Graphs can be directed:

**Definition 5** (Directed Set–Edge)**.** *Given two Set–Vertices,* $V^a$ *and* $V^b$*, with* $V^a \cap V^b = \emptyset$ *or* $V^a = V^b$*, a directed Set–Edge from* $V^a$ *to* $V^b$ *is a set of non repeated edges* $E[(V^a, V^b)] = \{e_1, e_2, \ldots, e_n\}$ *where each edge is an ordered pair of vertices* $e_i = (v_k^a \in V^a, v_l^b \in V^b)$.

**Definition 6** (Directed Set–Based Graph)**.** *A Directed Set–Based Graph is a pair* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *where*

- $\mathcal{V} = \{V^1, \ldots, V^n\}$ *is a set of disjoint set–vertices (i.e.,* $i \neq j \implies V^i \cap V^j = \emptyset$*).*

- $\mathcal{E} = \{E^1, \ldots, E^m\}$ *is a set of directed set–edges connecting set–vertices of* $\mathcal{V}$*, i.e.,* $E^i = E[(V^a, V^b)]$ *with* $V_a \in \mathcal{V}$ *and* $V_b \in \mathcal{V}$*. In addition, given two set edges* $E^i, E^j \in \mathcal{E}$ *with* $i \neq j$*, such that* $E^i = E[(V^a, V^b)]$ *and* $E^j = E[(V^c, V^d)]$*, then either* $V^a \neq V^c$ *or* $V^b \neq V^d$*. This is, two different set–edges in* $\mathcal{E}$ *cannot connect the same set–vertices with the same direction.*

An SB-Graph always defines an equivalent regular graph where the set of vertices and edges of the latter is the union of the set–vertices and set–edges of the former. Thus, a SB–Graph contains the same information than a regular graph. However, SB-Graphs can have a compact representation of that information provided that every set–edge and every set-vertex is defined by *intension*.

### *2.3. Stand-Alone QSS Solver*

The Stand-Alone QSS Solver [8] is a tool that was originally conceived to have an efficient implementation of Quantized State Systems (QSS) numerical integration algorithms [**?** ]. As these algorithms require the computation of incidence matrices, and the algorithms themselves are efficient in certain large scale models, the solver includes implementations of efficient automatic structure analysis methods in which this work is partially inspired.

The models in the Stand-Alone QSS Solver are written in a subset of the Modelica language [9] called *micro-Modelica* ($\mu$–Modelica). These models are parsed by the tool and translated into the C code that computes the right hand side of the corresponding ODE (including zero crossing functions and event handlers, since QSS algorithms are particularly efficient for discontinuous ODEs). The translation also includes the aforementioned structural analysis and the code produced allows to compute individual state derivatives and different incidence matrices. A remarkable feature of this translation is that it is performed preserving the arrays of the model definition and without expanding `for loop` statements.

Besides implementing QSS methods, the tool offers a front-end to classic ODE and DAE solvers like DASSL, CVODE, IDA, and DOPRI. For these methods, the model is also automatically translated into C code with all the necessary structures used by these solvers. In addition, using the theory developed in this work, the code for computing the symbolic Jacobian matrix is automatically generated by the tool (for CVODE-BDF and IDA).

### *2.4. Related Work*

While there is a large account of works dealing with automatic computation of sparse Jacobian matrices, to the best of our knowledge the problem of looking for a compact piece of code has not been considered so far. The literature on AD focus on having a code that minimizes the number of operations when the Jacobian is evaluated, but it does not consider the cost of producing that piece of code (including the time needed to compile it).

Among the different works on symbolic computation of Jacobian matrices, there are some that are specifically related to Modelica simulation [12, 13, 14]. We mention this ~~as this~~ since one of the main motivations of this work is related to expanding the abilities of Modelica tools to simulate large scale systems, a key problem faced nowadays by the community [15, 16, 17]

The algorithm proposed here is inspired by the problem of *causalization* of large systems of equations presented in [7] for which the concept of SB-Graphs was developed. There, algorithm of maximum matching and strongly connected components (Tarjan's) where developed in the context of SB-Graphs, which were constructed out of sets of equations written in Modelica language. A remarkable property of those algorithms was that, as in this work, their complexity was independent on the size of the arrays involved.

## 3. Main Results

This section presents the main result of the work, namely, the novel algorithm for producing the compact code for computing the sparse Jacobian. For simplicity, we introduce first in Section 3.1 a simple scalar version of the algorithm that does not take into account the presence of arrays of equations or variables. Then, in Section 3.2 we describe the SB-Graph representation of a large system like that of Eq.(4), and then in Section 3.3 we finally introduce the novel algorithm.

### *3.1. A scalar algorithm*

A preliminary algorithm for computing the sparse Jacobian matrix corresponding to the system of Eq.(1) is proposed next. Although it is not optimal and it includes some unnecessary steps and the graph representation of the model contains redundant vertices and edges, it can be easily extended for Set–Based Graphs.

The algorithm requires that the model is represented by a directed bipartite graph that is built based on the following rules:

- A vertex $X_i$ is associated to each state variable $x_i$.
- A vertex $A_i$ is associated to each algebraic variable $a_i$.
- A vertex $G_i$ is associated to each function $g_i(\cdot)$.
- A vertex $F_i$ is associated to each function $f_i(\cdot)$.
- An edge is directed from each vertex $A_i$ to the vertex $G_i$ (i.e., the function $g_i$ that computes $a_i$).
- If the state $x_j$ appears in the expression of $f_i$, then an edge is directed from $F_i$ to $X_j$. Similarly, if the algebraic variable $a_j$ appears in the expression of $f_i$ then an edge is directed from $F_i$ to $A_j$.
- If the state $x_j$ appears in the expression of $g_i$, then an edge is directed from $G_i$ to $X_j$. Similarly, if the algebraic variable $a_j$ appears in the expression of $g_i$ then an edge is directed from $G_i$ to $A_j$.

Based on this representation, each path that starts in a function vertex $F^i$ and finishes in a state vertex $X^j$ implies that there is a term that must be added to compute $\partial f^i/\partial x^j$. All the paths from each function to all the states can be found using a *Depth First Search* (DFS) on the graph, which can be exploited to compute also partial derivatives of the type $\partial f^i/\partial a^j$, $\partial g^i/\partial x^j$, and $\partial g^i/\partial a^j$ that are necessary to apply the chain rule whenever it is necessary.

Based on these ideas, we propose the following algorithm that produces the code to compute first all the partial derivatives and then, based on those partial derivatives, it produces the final code to compute the sparse Jacobian. The algorithm that performs the DFS also computes the set of states that the function depends on.

**Algorithm 1** Sparse Jacobian Code

1: **for** $i = 1 : n$ **do**
2: AddCode($F_i$,"$f_{[i]}$") ▷ DFS for computing $\partial f_i/\partial x$ and $F_i$.depStates
3: **end for**
4: **for** $i = 1 : n$ **do** ▷ Copy partial derivatives to sparse Jacobian Matrix
5: $col \leftarrow 0$

```
    for all x_j ∈ F_i.depStates do                ▷ Set of states that f_i depends on
        col ← col + 1                             ▷ col is the current column of the sparse Jacobian
        printCode "index([i],[col]) = [j]"        ▷ printCode writes into the final code
        printCode "J([i],[col]) = df_[i]dx_[j]"
    end for
    printCode "size([i]) = [col]"
end for
```

Here, the notation $[i]$ means that the expression is replaced in the string by the value of $i$.

The DFS that produces the code to compute the partial derivatives is performed by the following recursive function. This function also computes the set of states that the function depends on:

**Algorithm 2** Partial Derivative Computation

```
function AddCode(V,v)                             ▷ V is a function vertex, v is its name
    V.depStates=∅
    for all A_i ∈ Succ(V) do                      ▷ Algebraic variables that v directly depends on
        if A_i.visited==false then
            AddCode(G_i,"g_[i]")                  ▷ Recursive call for computing ∂g_i/∂x
            A_i.visited ← true
        end if
    end for
    for all A_i ∈ Succ(V) do                      ▷ Algebraic variables that v depends on
        printCode "aux = " + expr(∂v/∂a_i)        ▷ Code of the symbolic derivative (obtained by AD)
        for all x_j ∈ G_i.depStates do            ▷ States that g_i depends on
            printCode "d[v]dx_[j] = d[v]dx_[j] + aux · dg_[i]dx_[j]"   ▷ Chain Rule
        end for
        V.depStates ← V.depStates ∪ G_i.depStates   ▷ States that v indirectly depends on
    end for
    for all X_i ∈ Succ(V) do                      ▷ State variables that v directly depends on
        printCode "d[v]dx_[i] = d[v]dx_[i]" + expr(∂v/∂x_i)   ▷ Code of the symbolic derivative
        V.depStates ← V.depStates ∪ {x_i}
    end for
end function
```

The code produced assumes that the algebraic variables $a_i$ are all computed before invoking the function. It also assumes that all partial derivatives~~variables~~ are initialized at zero. Function expr$(\partial v/\partial x_i)$ computes the symbolic partial derivative of function $v$ using automatic differentiation.

In order to illustrate the way the algorithm works, we consider the following system

$$\begin{aligned} a_1 &= x_1^3 \\ a_2 &= a_1 + x_2 \\ \dot{x}_1 &= -a_1 + x_1 \\ \dot{x}_2 &= -a_2 + a_1^2 \end{aligned} \tag{5}$$

that can be represented by the graph of Figure 1

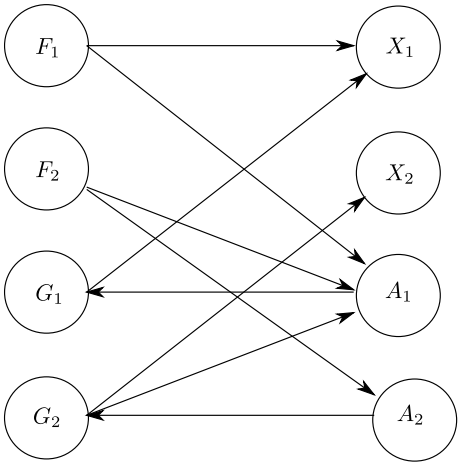


Figure 1: Bipartite directed graph corresponding to Eq. (5)

The piece of code produced by the algorithm and some of the intermediate steps it performs is listed below:

---

// Call AddCode($F_1$,"$f_1$"), Succ($F_1$)=$\{A_1, X_1\}$
// For successor $A_1$, call AddCode($G_1$,"$g_1$"), Succ($G_1$)=$\{X_1\}$
$dg_1dx_1 = dg_1dx_1 + 3 \cdot x_1^2$ // $\partial g_1/\partial x_1$
// $G_1$.depStates = $\{x_1\}$, end AddCode
// $A_1$.visited = true
$aux = -1$ // $\partial f_1/\partial a_1$
$df_1dx_1 = df_1dx_1 + aux \cdot dg_1dx_1$
// $F_1$.depStates = $F_1$.depStates $\cup$ $G_1$.depStates = $\{x_1\}$
$df_1dx_1 = df_1dx_1 + 1$ // $\partial f_1/\partial x_1 = 1$
// $F_1$.depStates = $\{x_1\}$, end AddCode
// Call AddCode($F_2$,"$f_2$"), Succ($F_2$)=$\{A_1, A_2\}$, $A_1$ was visited
// For successor $A_2$, call AddCode($G_2$,"$g_2$"), Succ($G_2$)=$\{A_1, X_2\}$
// For successor $A_1$
$aux = 1$ // $\partial g_2/\partial a_1$
$dg_2dx_1 = dg_2dx_1 + aux \cdot dg_1dx_1$
// $G_2$.depStates = $G_2$.depStates $\cup$ $G_1$.depStates = $\{x_1\}$
// For successor $X_2$
$dg_2dx_2 = dg_2dx_2 + 1$ // $\partial g_2/\partial x_2$
// $G_2$.depStates = $G_2$.depStates $\cup$ $\{x_2\}$ = $\{x_1, x_2\}$, end AddCode
// A_2.visited=true
$aux = 2 \cdot a_1$ // $\partial f_2/\partial a_1$
$df_2dx_1 = df_2dx_1 + aux \cdot dg_1dx_1$
// $F_2$.depStates = $F_2$.depStates $\cup$ $G_1$.depStates = $\{x_1\}$
$aux = -1$ // $\partial f_2/\partial a_2$
$df_2dx_1 = df_2dx_1 + aux \cdot dg_2dx_1$
$df_2dx_2 = df_2dx_2 + aux \cdot dg_2dx_2$
// $F_2$.depStates = $F_2$.depStates $\cup$ $G_2$.depStates = $\{x_1, x_2\}$
// End AddCode
$index(1,1) = 1$ //i=1, col=1, j=1
$J(1,1) = df_1dx_1$
$size(1) = 1$
$index(2,1) = 1$ //i=2, col=1, j=1

$J(2,1) = df_2 dx_1$
$index(2,2) = 2$ `//i=2, col=2, j=2`
$J(2,2) = df_2 dx_2$
$size(2) = 2$

---

The piece of code produced evaluates both the sparse Jacobian and its structure. Taking into account that the structure usually does not change during simulations, it is preferable to split the code so that the structure (i.e., *size* and *index* arrays) is computed in a separate piece of code. That could be easily done with a straightforward modification of the algorithm proposed.

### *3.2. Set-Based Graph Representation*

Given the ODE of Eq.(4), we could still apply the previous algorithm by rewriting the set of equations in the form of Eq.(1). However, by doing so, the number of vertices and edges in the graph and the length of the code produced will depend on the size of the arrays. If the size of the code is very large, then the compiler will take hours to compile. Moreover, beyond certain size, it will not be able to handle the code at all.

Thus, in order to overcome this problem, we propose first to represent the large system of equations of Eq.(4) using a set-based graph built according to the following rules:

- A set-vertex $X^k$ is associated to each state array $\mathbf{x^k}$.
- A set-vertex $A^k$ is associated to each algebraic array $\mathbf{a^k}$.
- A set-vertex $G^{k,l}$ is associated to each array of functions $\mathbf{g^{k,l}} = [\mathbf{g_1^{k,l}}, \mathbf{g_2^{k,l}}, \ldots]$.
- A set-vertex $F^{k,l}$ is associated to each array of functions $\mathbf{f^{k,l}} = [\mathbf{f_1^{k,l}}, \mathbf{f_2^{k,l}}, \ldots]$.
- Directed set-edges $E_G^{k,l}$ are directed from each set vertex $A^k$ to the set-vertices $G^{k,l}$, where $E_G^{k,l} = \bigcup_i \{(A^k_{i+s_g^{k,l}}, G_i^{k,l})\}$
- Given $k, m, l$, all the pairs $(F_i^{k,l}, X_j^m)$ such that $x_j^m$ appears in the expression of $f_i^{k,l}$ form a directed set-edge from $F^{k,l}$ to $X^m$, namely $E_{FX}^{k,l,m} = \bigcup_{i,j} \{(F_i^{k,l}, X_j^m)\}$.
- Similarly, given $k, m, l$, all the pairs $(F_i^{k,l}, A_j^m)$ such that $a_j^m$ appears in the expression of $f_i^{k,l}$ form a directed set-edge from $F^{k,l}$ to $A^m$, namely $E_{FA}^{k,l,m} = \bigcup_{i,j} \{(F_i^{k,l}, A_j^m)\}$.
- Given $k, m, l$, all the pairs $(G_i^{k,l}, X_j^m)$ such that $x_j^m$ appears in the expression of $g_i^{k,l}$ form a directed set-edge from $G^{k,l}$ to $X^m$, namely $E_{GX}^{k,l,m} = \bigcup_{i,j} \{(G_i^{k,l}, X_j^m)\}$.
- Similarly, given $k, m, l$, all the pairs $(G_i^{k,l}, A_j^m)$ such that $a_j^m$ appears in the expression of $g_i^{k,l}$ form a directed set-edge from $G^{k,l}$ to $A^m$, namely $E_{GA}^{k,l,m} = \bigcup_{i,j} \{(G_i^{k,l}, A_j^m)\}$.

The fact thats $f_i^{k,l}$ depends on $x_j^m$ and that it has the same expression for all $i$ implies that $j$ can be computed as a known function of $i$. This is, given $i$, we can compute which is the value of $j$ such that $x_j^m$ appears in the expression of $f_i^{k,l}$. This is true, at least, when $x_j^m$ appears only once in the expression of $f_i^{k,l}$. This index function is denoted as $j = \mathrm{map}_1[f^{k,l}, x^m](i)$.

If entries of $x^m$ appear more than once in the definition of $f_i^{k,l}$ (in expressions like $f_i^{k,l} = x_{i-1}^m - x_i^m$, for instance), then index functions $\text{map}_2[f^{k,l}, x^m]$, $\text{map}_3[f^{k,l}, x^m]$, etc. can be defined.

Thus, the set-edge $E_{FX}^{k,l,m}$ can be defined as

$$E_{FX}^{k,l,m} = \left( \bigcup_i \{(F_i^{k,l}, X^m_{\text{map}_1[f^{k,l},x^m](i)})\} \right) \cup \left( \bigcup_i \{(F_i^{k,l}, X^m_{\text{map}_2[f^{k,l},x^m](i)})\} \right) \cup \dots$$

Notice that this definition only requires to know the index functions $\text{map}_r$ and they do not depend on the size of the arrays involved.

The set-edges $E_{FA}^{k,l,m}$, $E_{GX}^{k,l,m}$, and $E_{GA}^{k,l,m}$ can be analogously defined and then all the set–edges can be defined by *intension* without any dependence on the size of the arrays involved.

### *3.3. Set-Based Graph Algorithm*

For the set–based graph described above, we propose an extension of Algorithms 1-2 to produce the code for the sparse Jacobian evaluation.

**Algorithm 3** Set–Based Graph Sparse Jacobian Code

```
1:  for i = 1 : n do
2:      for j = 1 : n_i do
3:          AddCode(F^{i,j}, "f^{[i],[j]}")                    ▷ Code for ∂f^{i,j}/∂x
4:      end for
5:  end for
6:  printCode "r = 0"                  ▷ Copy partial derivatives to sparse Jacobian matrix
7:  for i = 1 : n do
8:      for j = 1 : n_i do
9:          printCode "for row = r + 1 : r + [F^{i,j}.size]"
10:         printCode " for col = 1 : df^{[i,j]}dx.size(row − r)"
11:         printCode "   index(row, col) = df^{[i,j]}dx.index(row − r, col)"   ▷ Copy the structure
12:         printCode "   J(row, col) = df^{[i,j]}dx(row − r, col)"             ▷ Copy the values
13:         printCode " end for"
14:         printCode " size(row) = df^{[i,j]}dx.size(row − r)"
15:         printCode "end for"
16:         printCode "r = r + [F^{i,j}.size]"
17:     end for
18: end for
```

The algorithm copies the Jacobian matrix values and structure from the matrices that store the partial derivatives of each function $f^{i,j}$ on the right hand side of Eq.(4). These partial derivatives have also sparse representation and the code for its computation is produced by a recursive function that extends Algorithm 2.

The new recursive function, besides producing the code for computing the partial derivatives in sparse form, returns an array of dependences $F^{i,j}$.depStates($d$), with $d = 1, \dots, F^{i,j}$.numDeps. Each dependence is a subset of a set vertex~~edge~~, i.e., $F^{i,j}$.depStates($d$)$\subseteq X^k$ and it indicates that the components of function $f^{i,j}$ depends on the corresponding components of $x^k$. For each dependence, the recursive function also

computes a map $F^{i,j}.\text{Map}(d)$ that establishes the structure of this dependence, i.e., $F^{i,j}.\text{Map}(d)(l) = m$ implies that $f_l^{i,j}$ depends on $x_m^k$.

The dependence maps are built making use of the maps that define the different set edges $E_{FX}$, $E_{FA}$, etc. The states grouped in the same dependence set $F^{i,j}.\text{depStates}(d) \subseteq X^k$ have the properties that they belong to the same array ($x^k$) and that they are computed through the same map $F^{i,j}.\text{Map}(d)$. Thus, their contribution to the matrix $\partial f^{i,j}/\partial x$ can be computed using the same piece of code.

All the partial derivatives must be computed with respect to the global state vector $\mathbf{x} = [\mathbf{x^1}^T, \ldots, \mathbf{x^n}^T]^T$. Thus, the recursive function makes use of a state map (xMap) that maps the components of the different arrays into the components of the global state vector, i.e. $x_{\text{xMap}(k,l)} = x_l^k$. The construction of this map is straightforward and it is assumed to be known before the algorithm starts.

Based on these observations, the algorithm for producing the code that computes the partial derivatives in sparse representation is the following one:

**Algorithm 4** Set–Based Partial Derivative Computation

**function** AddCode($V$,$v$) ▷ $V$ is a function set-vertex, $v$ is its name
  $d \leftarrow 0$ ▷ Number of different state dependences of $v$
  **for all** $A^i \in \text{Succ}(V)$ **do** ▷ Algebraic variables that $v$ directly depends on
    **if** $A^i$.visited==false **then**
      **for all** $G^{i,j} \in \text{Succ}(A^i)$ **do** ▷ $g^{i,j}$ are functions that compute components of $a^i$
        AddCode($G^{i,j}$,"$g^{[i,j]}$") ▷ Code for computing $\partial g^{i,j}/\partial x$
      **end for**
      $A^i$.visited $\leftarrow$ true
    **end if**
  **end for**
  printCode "for $row = 1 : [V.size]$" ▷ Code for traversing all the components of $v$
  **for all** $A^i \in \text{Succ}(V)$ **do** ▷ Algebraic variables that $v$ directly depends on
    **for all** $\text{map}_m$ of $E_{VA}^i$ **do** ▷ Each map is a different dependence of $v$ on $a^i$
      printCode "$aux$ = " + $\partial v_l/\partial a^i_{\text{map}_m(l)}$ ▷ Code for symbolic partial derivative
      $\hat{A} \leftarrow \{a_l^i \in A^i : l = \text{map}_\text{m}(k), 1 \le k \le \text{size}(A^i)\}$
      **for all** $G^{i,j} \in \text{Succ}(A^i)$ **do** ▷ $g^{i,j}$ are functions that compute components of $a^i$
        **for** $d_g = 1 : G^{i,j}.\text{numDeps}$ **do** ▷ Traverse the dependences of $G^{i,j}$
          **if** $\hat{A} \cap \text{dom}(G^{i,j}.\text{Map}(d_g)) \neq \emptyset$ **then**
            nMap $\leftarrow [G^{i,j}.\text{Map}(d_g)$ - $s_g^{i,j}] \circ \text{map}_m$ ▷ Map from $v$ to $G^{i,j}.\text{depStates}(d_g)$
            $\hat{X} \leftarrow \{x_l^k \in G^{i,j}.\text{depStates}(d_g) \subseteq X^k : l = \text{nMap}(m), 1 \le m \le \text{size}(V)\}$ ▷ $\hat{X}$ contains only the states of $G^{i,j}.\text{depStates}(d_g)$ that can be reached from $V$ through nMap.
            **if** $\hat{X} \neq \emptyset$ **then**
              $d \leftarrow d + 1$ ▷ $\hat{X} \subseteq X^k$ is a new dependence for $V$
              $V.\text{Map}(d) \leftarrow$ nMap
              $V.\text{depStates}(d) \leftarrow \hat{X}$
              printCode "if $[nMap](row) \in [\hat{X}.indices]$ then" ▷ Check that $x^i_{nMap(row)} \in \hat{X}$
              printCode " $xind = xMap([k], [nMap](row))$" ▷ $v_{row}$ depends on global $x_{xind}$
              printCode " if $xind \notin d[v]dx, index(row, \cdot)$ then"

28: printCode ” $d[v]dx.size(row) = d[v]dx.size(row) + 1$” ▷ New column
29: printCode ” $col = d[v]dx.size(row)$”
30: printCode ” $d[v]dx.index(row, col) = xind$”
31: printCode ” else”
32: printCode ” $col = \text{pos}(xind \text{ in } d[v]dx.index(row, \cdot))$”
33: printCode ” end if”
34: printCode ”$rowg = [\text{map}_m](row) - [s_g^{i,j}]$”
35: printCode ”$colg = \text{pos}(xind \text{ in } dg^{[i,j]}dx.index(rowg, \cdot))$” ▷ We need $\partial g^{i,j}_{rowg}/\partial x_{xind}$ to apply the chain rule
36: printCode ” $d[v]dx(row, col) = d[v]dx(row, col) + aux \cdot dg^{i,j}dx(rowg, colg)$”
37: printCode ”end if”
38: **end if**
39: **end if**
40: **end for**
41: **end for**
42: **end for**
43: **end for**
44: **for all** $X^i \in \text{Succ}(V)$ **do** ▷ State arrays that $v$ directly depends on
45: **for all** $\text{map}_m$ of $E^i_{VX}$ **do** ▷ Each map defines a different dependence
46: $\hat{X} = \{x^i_l \in X^i : l = \text{map}_m(j), 1 \leq j \leq \text{size}(V)\}$ ▷ Image of $V$ through $\text{map}_m$
47: $d \leftarrow d + 1$
48: $V.\text{depStates}(d) \leftarrow \hat{X}$ ▷ New dependence
49: $V.\text{Map}(d) \leftarrow \text{map}_m$
50: printCode ”if $[\text{map}_m](row) \in [\hat{X}.indices]$ then” ▷ Check that $x^i_{map_m(row)} \in \hat{X}$
51: printCode ” $xind = xMap([i], [\text{map}_m](row))$” ▷ $v_{row}$ depends on $x_{xind}$
52: printCode ” if $xind \notin d[v]dx, index(row, \cdot)$ then”
53: printCode ” $d[v]dx.size(row) = d[v]dx.size(row) + 1$” ▷ New column in the current row
54: printCode ” $col = d[v]dx.size(row)$”
55: printCode ” $d[v]dx.index(row, col) = xind$”
56: printCode ” else”
57: printCode ” $col = \text{pos}(xind \text{ in } d[v]dx.index(row, \cdot))$”
58: printCode ” end if”
59: printCode ”$aux =$ ” + $\partial v_l/\partial x^i_{\text{map}_m(l)}$ ▷ Code for the symbolic derivative
60: printCode ” $d[v]dx(row, col) = d[v]dx(row, col) + aux$”
61: printCode ”end if”
62: **end for**
63: **end for**
64: $V.\text{numDeps} = d$
65: printCode ”end for”
66: **end function**

This algorithm, just like the scalar algorithm (Algorithm 2), first perform a DFS in order to produce the code for computing all the partial derivatives of functions $g^{i,j}$ that are needed to apply the chain rule. Then, it produces the code for computing the partial derivatives of the current function $\partial v/\partial x$.

The main difference with the scalar algorithm is that now the state dependences of each function

($V$.depStates) are not individual states but are subsets of the state arrays. In addition, for each state dependence $\hat{X} \subseteq X^k$, the algorithm builds an index map $V.Map$ between the components of the function $V$ and the components of the state arrays $X^k$.

These differences are also reflected in the written code. First, the new algorithm writes a `for loop` sentence that traverses the different components of $v$ which are translated into rows of the sparse matrix representing $\partial v/\partial x$. In addition, during the traversal of the components of $v$, the code uses the maps $V.Map(d)$ to relate the rows with the components of the $X^k$ and checks that the result of applying the map leads effectively to components of each state dependence $\hat{X} \subseteq X^k$. It could happen that only some components of $V$ effectively depend on components of $\hat{X}$.

This algorithm can be improved in several ways. Firstly, it does not take into account the situation in which an algebraic variable $a^i_j$ depends on another algebraic variable $a^i_k$ that belongs to the same array (with $k < j$ to preserve causality). This case would be detected at line 16 when $G^{i,j} = V$ (i.e., when a function $G^{i,j}$ is a successor of itself). Although for reasons of space this special case is not treated here, the right code can be generated using the information about the recursive use of the maps $\mathrm{map}_m[g^{i,j}a^i]$.

Another case that is not properly explained is the presence of expressions like $a^i_k = \sum_j x^m_j$, where a single algebraic (or state derivative) variable depends on several components of the same array. As presented in the algorithm above, that case would be represented by the use of several maps (up to the the number of terms in the sum). However, it can be simplified by generalizing the set–edge representation so that they have not only an image map, but also a domain map, i.e.,

$$E^{k,l,m}_{FX} = \left( \bigcup_i \{ (F^{k,l}_{\mathrm{map}_{dom}[f^{k,l},x^m](i)}, X^m_{\mathrm{map}_{im}[f^{k,l},x^m](i)}) \} \right)$$

so that a single equation depending on several states can be described by a single map. Then, this compact information can be used to produce a compact piece of code (including iterations over the states at the right hand side) for the Jacobian matrix.

Also, as in the previous example, the algorithm can be straightforwardly modified so that it splits the code corresponding to the Jacobian values and its structure.

In order to illustrate the way this algorithm works, we consider the following model, which represent the equations of a nonlinear RC transmission line of arbitrary size (given by the parameter $N$) and with an inductive load.

$$\begin{aligned}
a^1_1 &= \frac{(10 - x^1_1)^3}{R} = g^{1,1}_1(\cdot), \\
a^1_{i+1} &= \frac{(x^1_i - x^1_{i+1})^3}{R} = g^{1,2}_i(\cdot), \quad i = 1, \ldots, N-1 \\
\dot{x}^1_i &= \frac{a^1_i - a^1_{i+1}}{C} = f^{1,1}_i(\cdot), \quad i = 1, \ldots, N-1 \\
\dot{x}^1_N &= \frac{a^1_N - x^2_1}{C} = f^{1,2}_1(\cdot), \\
\dot{x}^2_1 &= \frac{x^1_N}{L} = f^{2,1}_1(\cdot).
\end{aligned} \tag{6}$$

The set–graph representation for this model is depicted in Figure 2. It contains three set vertices for the variables $X^1$, $X^2$, $A^1$, and five set vertices for the functions, $F^{1,1}$, $F^{1,2}$, $F^{2,1}$, $G^{1,1}$, and $G^{1,2}$.

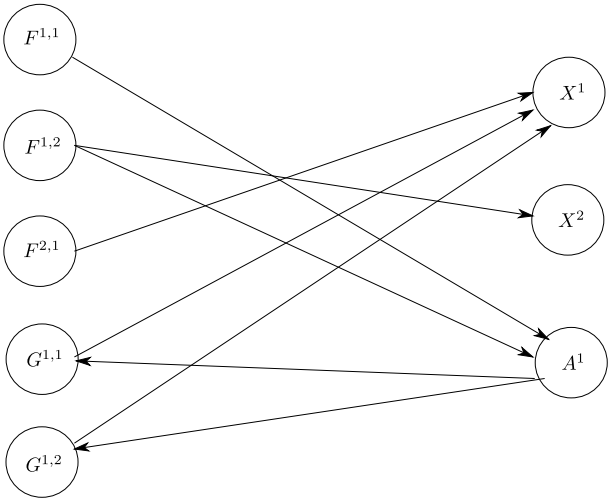


Figure 2: Bipartite directed set-based graph corresponding to Eq. (6)

There are eight set edges characterized by the following maps: $\text{map}_1[f^{1,1}, a^1](i) = i$, $\text{map}_2[f^{1,1}, a^1](i) = i + 1$, $\text{map}_1[f^{1,2}, a^1](1) = N$, $\text{map}_1[f^{1,2}, x^2](1) = 1$, $\text{map}_1[f^{2,1}, x^1](1) = 1$, $\text{map}_1[g^{1,1}, x^1](1) = 1$, $\text{map}_1[g^{1,2}, x^1](i) = i$, $\text{map}_2[g^{1,2}, x^1](i) = i + 1$, $\text{map}_1[a^1, g^{1,1}](1) = 1$, and $\text{map}_1[a^1, g^{1,2}](i) = i + 1$.

The steps performed and the code produced by the algorithm on this example is listed in Appendix Appendix A.

## 4. Implementation

A variant of Algorithms 3-4 described in the previous section was implemented as part of the Stand Alone QSS Solver [8]. The implementation has the following features:

- The models are described in a subset of Modelica language called $\mu$-Modelica. This sub-language allows describing systems in ODE form and it has support for multi-dimensional arrays.
- The Modelica code is parsed and a Set-Based Graph is built based on the rules described in Section 3.2. Set–Based Graph are implemented by extending features of~~based on~~ the Boost library [18].
- The different sets and subset of vertices involved in the algorithm are represented by unions of *Multidimensional Interval Arrays* (MDI). Each MDI is defined by three integer numbers in each dimension (initial index, final index and step). That way, the representation complexity is independent on the size of the arrays.
- There are implementations of operations like union, intersection and mutual exclusion between sets. These operation also produce sets of MDIs with a complexity that is independent on the size of the arrays.

- The index maps that define the set-edges are limited to linear affine functions between MDIs indices. Thus, each map is characterized by two integer numbers in each dimension. Also, it is simple to apply the composition of several maps as it is required by the algorithm.
- There are also operations that compute the image map of sets represented by MDIs that do not depend on the size of the MDIs.
- Making use of these features, the algorithm for establishing the sets of dependences of a function ($v$.depStates) is almost identical to that described by Algorithm 4.
- The code generation, however, has some differences. Here, instead of computing a sparse matrix for the partial derivative of each function (including those that define algebraic variables), the algorithm directly produces the code for computing the sparse Jacobian at once. That way, a more compact piece of code is produced, yet the algorithm is more complex.
- The algebraic differentiation needed in lines 14 and 59 of Algorithm 4 is performed using GiNaC [19].
- The final piece of code produced is written in plain C language.

## 5. Examples

In order to evaluate the code produced by the proposed algorithm, we present two examples where we measure the compilation time of the generated models for different problem sizes and the simulation time for the sparse and dense Jacobian representations. Additionaly, we compare the compilation time of the expanded models for different sizes (without `for` statements).

We first revisit the RC transmission line example presented previously and then we evaluate a two dimensional advection diffusion reaction ($ADR$) model. In both examples we use the following setup:

- The reported results were obtained using an Intel(R) Core(TM) $i7-8650U$@$1.90GHz$ CPU with 24 GB of RAM memory. We used QSS Solver version ADD TAG and CVODE_BDF solver on Ubuntu OS.
- For each experiment, 10% of the total output variables were randomly selected to compute the error.
- The reported error was computed as the average of the selected output variables using the RMSE (Root Mean Square Error) metric.
- The ground truth values were obtained using DASSL over 5000 sampled points with the following tolerances: $tol_{rel} = 1 \cdot 10^{-10}$ and $tol_{abs} = 1 \cdot 10^{-10}$.

### *5.1. RC Transmission Line*

This model defined previously in Eq.(6) represents a nonlinear RC transmission line of arbitrary size, given by the parameter $N$ and with an inductive load. Additionaly, we set the rest of the model

parameters to $R = 1$, $C = 1$, $L = 1$ with tolerances set to $tol_{rel} = 1 \cdot 10^{-4}$ and $tol_{abs} = 1 \cdot 10^{-4}$. Finally, the initial condiciotns are $x_i^1 = 1$ for $i = 1, \ldots, N$ and $x_1^2 = 0$.

$$
\begin{aligned}
a_1^1 &= \frac{(10 - x_1^1)^3}{R} \\
a_{i+1}^1 &= \frac{(x_i^1 - x_{i+1}^1)^3}{R} \quad i = 1, \ldots, N-1 \\
\dot{x}_i^1 &= \frac{a_i^1 - a_{i+1}^1}{C} \quad i = 1, \ldots, N-1 \\
\dot{x}_N^1 &= \frac{a_N^1 - x_1^2}{C} \\
\dot{x}_1^2 &= \frac{x_N^1}{L}
\end{aligned}
\tag{7}
$$

Table 1 shows the simulation times and the errors obtained for different sizes of the $N$ parameter with CVODE_BDF solver configured to use the sparse and dense representation he Jacobian. As expected the error in both cases respect the selected tolerances with gains up to 33 times when $N = 20000$ when using the sparse Jacobian representation due to the overhead of computing the dense Jacobian representation.

| **Size** | **Sparse Simulation Time [msec]** | **Error** | **Dense Simulation Time [msec]** | **Error** |
|---|---|---|---|---|
| 100 | 14 | $4.54 \cdot 10^{-4}$ | 38 | $6.17 \cdot 10^{-4}$ |
| 200 | 21 | $9.78 \cdot 10^{-5}$ | 104 | $9.44 \cdot 10^{-5}$ |
| 500 | 70 | $3.70 \cdot 10^{-4}$ | 586 | $2.36 \cdot 10^{-4}$ |
| 1000 | 162 | $6.53 \cdot 10^{-5}$ | 1062 | $1.21 \cdot 10^{-4}$ |
| 2000 | 473 | $2.24 \cdot 10^{-4}$ | 11335 | $2.49 \cdot 10^{-4}$ |
| 5000 | 2584 | $1.08 \cdot 10^{-4}$ | 77289 | $9.17 \cdot 10^{-5}$ |
| 10000 | 9400 | $8.27 \cdot 10^{-5}$ | 298457 | $6.84 \cdot 10^{-5}$ |
| 20000 | 33260 | $3.08 \cdot 10^{-5}$ | $1.11 \cdot 10^{6}$ | $4.54 \cdot 10^{-5}$ |

Table 1: RC Transmission Line system results with different size of parameter $N$

The average compilation time for all models is 498 milliseconds and also as expected it remains constant as the size of the problem grows, on the other hand Figure 3 shows how the compilation time grows for different values of the $N$ when for loops are expanded.[1]

[1] Appendix A contains the full code generated for this example.

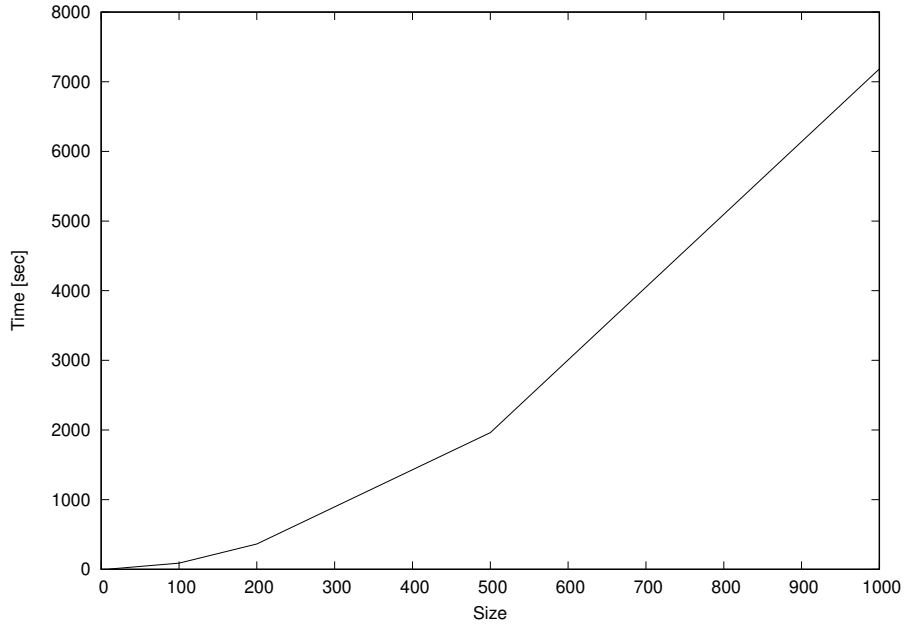


Figure 3: RC Transmission Line with for loops expanded compilation time.

*5.2. 2D Advection-Diffusion-Reaction (ADR) problem*

The following set of ODEs corresponds to the MOL (*Method Of Lines*) discretization of a 2D *ADR* problem:

$$
\begin{aligned}
\frac{\mathrm{d}u_{1,1}}{\mathrm{d}t} &= -a_x \cdot \frac{u_{1,1}}{\Delta x} + a_y \cdot \frac{u_{1,1}}{\Delta y} + r \cdot (u_{1,1}^2 - u_{1,1}^3) \\
\frac{\mathrm{d}u_{i,1}}{\mathrm{d}t} &= -a_x \cdot \frac{u_{i,1}}{\Delta x} + a_y \cdot \frac{(u_{i,1} - u_{i-1,1})}{\Delta y} + r \cdot (u_{i,1}^2 - u_{i,1}^3) \quad i = 2 \ldots, N \\
\frac{\mathrm{d}u_{1,j}}{\mathrm{d}t} &= -a_x \cdot \frac{(u_{1,j} - u_{1,j-1})}{\Delta x} + a_y \cdot \frac{u_{1,j}}{\Delta y} + r \cdot (u_{1,j}^2 - u_{1,j}^3) \quad j = 2, \ldots, N \\
\frac{\mathrm{d}u_{i,j}}{\mathrm{d}t} &= -a_x \cdot \frac{(u_{i,j} - u_{i,j-1})}{\Delta x} + a_y \cdot \frac{(u_{i,j} - u_{i-1,})}{\Delta y} + r \cdot (u_{i,j}^2 - u_{i,j}^3) \quad i = 2, \ldots, N \quad j = 2, \ldots, N
\end{aligned}
\tag{8}
$$

where $N$ is the number of grid points, and we consider parameters $a_x$, $a_y$, $r$, $\Delta x$ and $\Delta y$ to be equal to 1, and we set the initial conditions $u_{1,1} = 1$ and $u_{i,j} = 0$ for $i = 2, \ldots, N \quad j = 2, \ldots, N$.

The results obtained for the different Jacobian representation are shown in Table 2 where we can see that in this case using the sparse Jacobian impementation is 375 times faster than the dense representation for a grid of size 150. Again, as expected the error bounds are respected in both cases.

The average compilation time in this case is 404 milliseconds for all problem sizes, the compilation time performance for the expanded `for` loop models is shown in Figure 4.

| **Size** | **Sparse Simulation Time [msec]** | **Error** | **Dense Simulation Time [msec]** | **Error** |
|---|---|---|---|---|
| 10 | 2 | $4.91 \cdot 10^{-5}$ | 4 | $4.91 \cdot 10^{-5}$ |
| 20 | 6 | $4.57 \cdot 10^{-5}$ | 34 | $4.58.2 \cdot 10^{-5}$ |
| 30 | 14 | $6.12 \cdot 10^{-5}$ | 196 | $6.11 \cdot 10^{-5}$ |
| 40 | 25 | $1.04 \cdot 10^{-4}$ | 662 | $1.03 \cdot 10^{-4}$ |
| 50 | 39 | $1.56 \cdot 10^{-4}$ | 1898 | $1.56 \cdot 10^{-4}$ |
| 60 | 65 | $1.21 \cdot 10^{-4}$ | 4127 | $1.20 \cdot 10^{-4}$ |
| 70 | 98 | $1.34 \cdot 10^{-4}$ | 8689 | $1.33 \cdot 10^{-4}$ |
| 80 | 125 | $1.47 \cdot 10^{-4}$ | 17216 | $1.46 \cdot 10^{-4}$ |
| 90 | 184 | $2.20 \cdot 10^{-4}$ | 22748 | $2.22 \cdot 10^{-4}$ |
| 100 | 214 | $2.26 \cdot 10^{-4}$ | 32644 | $2.26 \cdot 10^{-4}$ |
| 150 | 555 | $2.15 \cdot 10^{-4}$ | 208464 | $2.16 \cdot 10^{-4}$ |

Table 2: 2D ADR results for different grid size paramter $N$

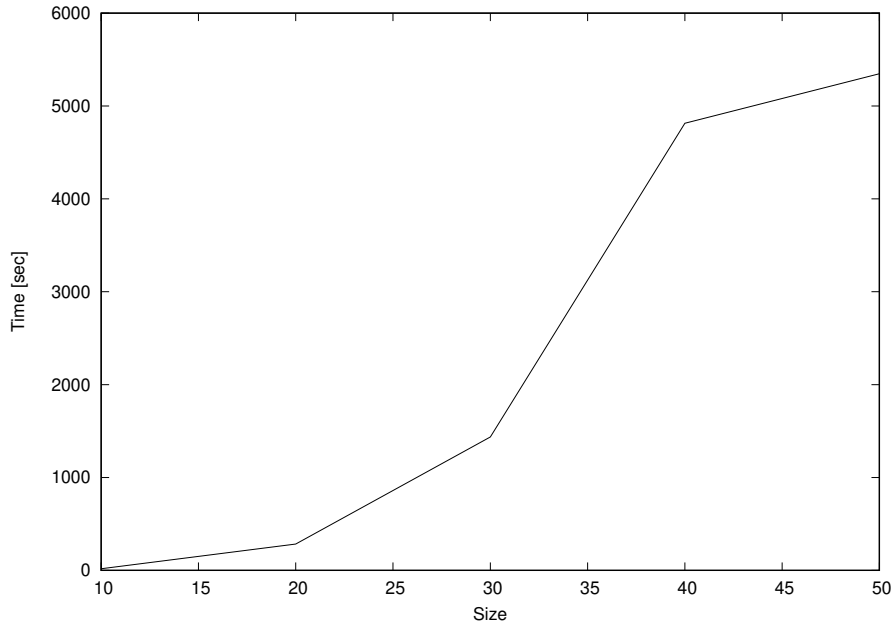


Figure 4: 2D ADR model with for loops expanded compilation time.

## 6. Conclusions and Future Research

We presented a novel algorithm that automatically computes a compact sparse symbolical Jacobian matrices representation for a given model, in particular large scale. To that goal, the algorithm takes

advantage of the compact *Set–Based Graph* representation so that the computational cost does not depend on the size of the model. We implemented the proposed approach in the Stand Alone QSS Solver and presented two simple examples that shows the advantages of the proposed algorithm. We are currently working on application of the *Set–Based Graph* compact representation of `for` loop statements to different stages of the compilation of large scale models.


## Funding

This work was partially funded by grant PICT–2017 2436 (ANPCYT).


## References


[1] F. E. Cellier, E. Kofman, Continuous System Simulation, Springer, New York, 2006.

[2] S. D. Cohen, A. C. Hindmarsh, P. F. Dubois, Cvode, a stiff/nonstiff ode solver in c, Computers in physics 10 (2) (1996) 138–143.

[3] L. A. Nejad, A comparison of stiff ode solvers for astrochemical kinetics problems, Astrophysics and Space Science 299 (1) (2005) 1–29.

[4] S. A. Forth, M. Tadjouddine, J. D. Pryce, J. K. Reid, Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding, ACM Transactions on Mathematical Software (TOMS) 30 (3) (2004) 266–299.

[5] U. Naumann, The art of differentiating computer programs: an introduction to algorithmic differentiation, Vol. 24, Siam, 2012.

[6] A. Elsheikh, An equation-based algorithmic differentiation technique for differential algebraic equations, Journal of Computational and Applied Mathematics 281 (2015) 135–151.

[7] P. Zimmermann, J. Fernández, E. Kofman, Set-based graph methods for fast equation sorting in large dae systems, in: Proceedings of the 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools, 2019, pp. 45–54.

[8] J. Fernández, E. Kofman, A Stand-alone Quantized State System Solver for Continuous System Simulation, Simulation 90 (7) (2014) 782–799.

[9] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: a Cyber-Physical Approach", Wiley-IEEE Press, 2015.

[10] B. M. Averick, J. J. Moré, C. H. Bischof, A. Carle, A. Griewank, Computing large sparse jacobian matrices using automatic differentiation, SIAM Journal on Scientific Computing 15 (2) (1994) 285–294.

[11] E. Varnik, Exploitation of structural sparsity in algorithmic differentiation., Ph.D. thesis, RWTH Aachen University (2011).

[12] W. Braun, L. Ochel, B. Bachmann, Symbolically derived jacobians using automatic differentiation-enhancement of the openmodelica compiler, in: Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany, no. 063, Linköping University Electronic Press, 2011, pp. 495–501.

[13] J. Åkesson, W. Braun, P. Lindholm, B. Bachmann, Generation of sparse jacobians for the function mock-up interface 2.0, in: Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany, no. 076, Linköping University Electronic Press, 2012, pp. 185–196.

[14] W. Braun, K. Kulshreshtha, R. Franke, A. Walther, B. Bachmann, Towards adjoint and directional derivatives in fmi utilizing adol-c within openmodelica, in: Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017, no. 132, Linköping University Electronic Press, 2017, pp. 363–366.

[15] F. Casella, Simulation of large-scale models in modelica: State of the art and future perspectives, in: 11th International Modelica Conference, 2015, pp. 459–468.

[16] W. Braun, F. Casella, B. Bachmann, et al., Solving large-scale modelica models: new approaches and experimental results using openmodelica, in: 12 International Modelica Conference, Linkoping University Electronic Press, 2017, pp. 557–563.

[17] G. Schweiger, H. Nilsson, J. Schoeggl, W. Birk, A. Posch, Modeling and simulation of large-scale systems: A systematic comparison of modeling paradigms, Applied Mathematics and Computation 365 (2020) 124713.

[18] J. Siek, A. Lumsdaine, L.-Q. Lee, The boost graph library: user guide and reference manual, Addison-Wesley, 2002.

[19] C. Bauer, A. Frink, R. Kreckel, Introduction to the GiNaC framework for symbolic computation within the C++ programming language, Journal of Symbolic Computation 33 (1) (2002) 1–12.

## Appendix A. Code generation for System (6)

```
// call AddCode(F^{1,1},"f^{1,1}"), the only successor is A^1 (not visited).
// call AddCode(G^{1,1},"g^{1,1}"), the only successor is X^1
for row = 1 : 1
  // the only map is map_m = map_1[g^{1,1}, x^1](1) = 1
  // G^{1,1}.depStates(1)={x_1^1}, G^{1,1}.Map(1)=map_1[g^{1,1}, x^1]
  if 1 ≤ row ≤ 1 then // because map_m(row) = row and X̂.indices=1
    xind=xMap(1,row) // because map_m(row) = row
    if xind ∉ dg^{1,1}dx.index(row, ·) then
      dg^{1,1}dx.size(row) = dg^{1,1}dx.size(row) + 1
      col = dg^{1,1}dx.size(row)
      dg^{1,1}dx.index(row, col) = xind
    else
      col = pos(xind in dg^{1,1}dx.index(row, ·))
    end if
    dg^{1,1}dx(row, col) = dg^{1,1}dx(row, col) − 3/R · (10 − x_{row}^1)^2 //∂g_i^{1,1}/∂x_i^1
  end if
  //G^{1,1}.numDeps=1
```

```
end for
// end AddCode(G^{1,1},g^{1,1}), back to AddCode(F^{1,1},"f^{1,1}")
// call AddCode(G^{1,2},"g^{1,2}"), the only successor is X^1
for row = 1 : N-1 //size of G^{1,2}
  // there are two maps: map_1[g^{1,2},x^1](i) = i and map_2[g^{1,2},x^1](i) = i+1
  // G^{1,2}.depStates(1) = X̂ = {x_i^1 | 1 ≤ i ≤ N-1}, G^{1,2}.Map(1)=map_m=map_1[g^{1,2},x^1](i) = i
  if 1 ≤ row ≤ N-1 then // because map_m(row) = row and X̂.indices = [1,...,N-1]
    xind=xMap(1,row) //because map_m(row) = row
    if xind ∉ dg^{1,2}dx.index(row,·) then
      dg^{1,2}dx.size(row) = dg^{1,2}dx.size(row) + 1
      col = dg^{1,2}dx.size(row)
      dg^{1,2}dx.index(row,col) = xind
    else
      col = pos(xind in dg^{1,2}dx.index(row,·))
    end if
    dg^{1,2}dx(row,col) = dg^{1,2}dx(row,col) + 3/R · (x^1_{row} - x^1_{row+1})^2 //∂g_i^{1,2}/∂x_i^1
  end if
  // G^{1,2}.depStates(2) = X̂ = {x_i^1 | 2 ≤ i ≤ N}, G^{1,2}.Map(2)=map_2[g^{1,2},x^1](i) = i+1
  if 2 ≤ row+1 ≤ N then //because map_2[g^{1,2},x^1](row) = row+1 and X̂.indices = [2,...,N]
    xind=xMap(1,row+1)
    if xind ∉ dg^{1,2}dx.index(row,·) then
      dg^{1,2}dx.size(row) = dg^{1,2}dx.size(row) + 1
      col = dg^{1,2}dx.size(row)
      dg^{1,2}dx.index(row,col) = xind
    else
      col = pos(xind in dg^{1,2}dx.index(row,·))
    end if
    dg^{1,2}dx(row,col) = dg^{1,2}dx(row,col) - 3/R · (x^1_{row} - x^1_{row+1})^2 //∂g_i^{1,2}/∂x_{i+1}^1
  end if
  // G^{1,2}.numDeps = 2
end for
// end AddCode(G^{1,2},g^{1,2}), back to AddCode(F^{1,1},"f^{1,1}")
for row = 1 : N-1 // size of F^{1,1}
  // E_{FA}^{1,1,1} contains two maps, take first map_m = map_1[f^{1,1},a^1](i) = i
  aux = 1/C //  ∂f_i^{1,1}/∂a_i^1 in map_1
  // G^{1,·} has two components
  // G^{1,1}.numDeps is 1 (it has one map map_1[g^{1,1},x^1](1) = 1)
  // nMap = G^{1,1}.Map(1) ∘ map_m ⟹ nMap(1) = 1
  // G^{1,1}.depStates(1) = {x_1^1}, X̂ = {x_1^1}
  // F^{1,1}.Map(1) = nMap, F^{1,1}.depStates(1) = {x_1^1}
  if 1 ≤ row ≤ 1 then //nMap(row)=row, X̂.indices = {1}
    xind = xMap(1,row) //nMap(row)=row
    if xind ∉ df^{1,1}dx,index(row,·) then
      df^{1,1}dx.size(row) = df^{1,1}dx.size(row) + 1
      col = df^{1,1}dx.size(row)
      df^{1,1}dx.index(row,col) = xind
    else
      col = pos(xind in df^{1,1}dx.index(row,·))
    end if
          rowg = row
    colg = pos(xind in dg^{1,1}dx.index(rowg,·))
    df^{1,1}dx(row,col) = df^{1,1}dx(row,col) + aux · dg^{1,1}dx(rowg,colg)
  end if
  // G^{1,2}.numDeps is 2
  // nMap = [G^{1,2}.Map(1) - s_g^{1,2}] ∘ map_m ⟹ nMap(i) = i-1
  // G^{1,2}.depStates(1) = {x_l^1 | 1 ≤ l ≤ N-1}, X̂ = {x_l^1 | 1 ≤ l ≤ N-2}
```

```
// F^{1,1}.Map(2) = nMap, F^{1,1}.depStates(2) = {x_l^1 | 1 ≤ l ≤ N − 2}
if 1 ≤ row − 1 ≤ N − 2 then //nMap(row)=row-1, X̂ = {x_l^1 | 1 ≤ l ≤ N − 2}.
  xind = xMap(1, row − 1) //nMap(row)=row-1
  if xind ∉ df^{1,1}dx, index(row, ·) then
    df^{1,1}dx.size(row) = df^{1,1}dx.size(row) + 1
    col = df^{1,1}dx.size(row)
    df^{1,1}dx.index(row, col) = xind
  else
    col = pos(xind in df^{1,1}dx.index(row, ·))
  end if
  rowg = row − 1
  colg = pos(xind in dg^{1,2}dx.index(rowg, ·))
  df^{1,1}dx(row, col) = df^{1,1}dx(row, col) + aux · dg^{1,2}dx(rowg, colg)
end if
// second dependence: nMap = [G^{1,2}.Map(2) − s_g^{1,2}] ∘ map_m ⟹ nMap(i) = i
// G^{1,2}.depStates(2) = {x_l^1 | 2 ≤ l ≤ N}, X̂ = {x_l^1 | 2 ≤ l ≤ N}
// F^{1,1}.Map(3) = nMap, F^{1,1}.depStates(3) = {x_l^1 | 2 ≤ l ≤ N}
if 2 ≤ row ≤ N then
  xind = xMap(1, row) //nMap(row)=row
  if xind ∉ df^{1,1}dx, index(row, ·) then
    df^{1,1}dx.size(row) = df^{1,1}dx.size(row) + 1
    col = df^{1,1}dx.size(row)
    df^{1,1}dx.index(row, col) = xind
  else
    col = pos(xind in df^{1,1}dx.index(row, ·))
  end if
        rowg = row − 1
  colg = pos(xind in dg^{1,2}dx.index(rowg, ·))
  df^{1,1}dx(row, col) = df^{1,1}dx(row, col) + aux · dg^{1,2}dx(rowg, colg)
end if
// E_{FA}^{1,1,1} contains two maps, take now map_m = map_2[f^{1,1}, a^1](i) = i + 1
aux = -1/C //  ∂f_i^{1,1}/∂a_{i+1}^1
// Applying map_m yields Â = {a_l^1 | 2 ≤ l ≤ N}
// G^{1,·} has two components
// G^{1,1}.numDeps is 1 (it has one map map_1[g^{1,1}, x^1](1) = 1)
// Given that Â ∩ dom(G^{1,1}.Map(1)) = ∅
// there's no dependence on this path, so continue with the following vertex.
// G^{1,2}.numDeps is 2
// nMap = [G^{1,2}.Map(1) − s_g^{1,2}] ∘ map_m ⟹ nMap(i) = i
// G^{1,2}.depStates(1) = {x_l^1 | 1 ≤ l ≤ N − 1}, X̂ = {x_l^1 | 1 ≤ l ≤ N − 1}
// F^{1,1}.Map(4) = nMap, F^{1,1}.depStates(4) = {x_l^1 | 1 ≤ l ≤ N − 1}
if 1 ≤ row ≤ N − 1 then
  xind = xMap(1, row) //nMap(row)=row
  if xind ∉ df^{1,1}dx, index(row, ·) then
    df^{1,1}dx.size(row) = df^{1,1}dx.size(row) + 1
    col = df^{1,1}dx.size(row)
    df^{1,1}dx.index(row, col) = xind
  else
    col = pos(xind in df^{1,1}dx.index(row, ·))
  end if
  rowg = row
  colg = pos(xind in dg^{1,2}dx.index(rowg, ·))
  df^{1,1}dx(row, col) = df^{1,1}dx(row, col) + aux · dg^{1,2}dx(rowg, colg)
end if
// second dependence: nMap = [G^{1,2}.Map(2) − s_g^{1,2}] ∘ map_m ⟹ nMap(i) = i + 1
// G^{1,2}.depStates(2) = {x_l^1 | 2 ≤ l ≤ N}, X̂ = {x_l^1 | 2 ≤ l ≤ N}
```

```
  // F^{1,1}.Map(5) = nMap, F^{1,1}.depStates(5) = {x_l^1 | 2 ≤ l ≤ N}
  if 2 ≤ row + 1 ≤ N then
    xind = xMap(1, row + 1) //nMap(row)=row+1
    if xind ∉ df^{1,1}dx, index(row, ·) then
      df^{1,1}dx.size(row) = df^{1,1}dx.size(row) + 1
      col = df^{1,1}dx.size(row)
      df^{1,1}dx.index(row, col) = xind
    else
      col = pos(xind in df^{1,1}dx.index(row, ·))
    end if
    rowg = row
          colg = pos(xind in dg^{1,2}dx.index(rowg, ·))
    df^{1,1}dx(row, col) = df^{1,1}dx(row, col) + aux · dg^{1,2}dx(rowg, colg)
  end if
  // F^{1,1}.numDeps=5
end for
// end AddCode(F^{1,1}, g^{1,1}), back to main algorithm
// call AddCode(F^{1,2}, "f^{1,2}"), the only successor is A^1 (already visited).
for row = 1 : 1 // size of F^{1,2}
  // E_{FA}^{1,2,1} contains one map map_m = map_1[f^{1,2}, a^1](i) = N
  aux = 1/C //  ∂f_i^{1,2}/∂a_N^1
  // In this case map_m yields Â = {a_N^1}
  // G^{1,·} has two components
  // G^{1,1}.numDeps is 1 (it has one map map_1[g^{1,1}, x^1](1) = 1)
  // Again, given that Â ∩ dom(G^{1,1}.Map(1)) = ∅
  // there's no dependence on this path, so continue with the following vertex.
  // G^{1,2}.numDeps is 2
  // nMap = [G^{1,2}.Map(1) − s_g^{1,2}] ∘ map_m ⟹ nMap(i) = N − 1
  // G^{1,2}.depStates(1) = {x_l^1 | 1 ≤ l ≤ N − 1}, X̂ = {x_{N−1}^1}
  // F^{1,2}.Map(1) = nMap, F^{1,2}.depStates(1) = {x_{N−1}^1}
  if N − 1 ≤ N − 1 ≤ N − 1 then //nMap(i)=N-1,
    xind = xMap(1, N − 1) //nMap(1)=N-1
    if xind ∉ df^{1,2}dx, index(row, ·) then
      df^{1,2}dx.size(row) = df^{1,2}dx.size(row) + 1
      col = df^{1,1}dx.size(row)
      df^{1,2}dx.index(row, col) = xind
    else
      col = pos(xind in df^{1,2}dx.index(row, ·))
    end if
    rowg = row
    colg = pos(xind in dg^{1,2}dx.index(rowg, ·))
    df^{1,2}dx(row, col) = df^{1,1}dx(row, col) + aux · dg^{1,2}dx(rowg, colg)
  end if
  // second dependence: nMap = [G^{1,2}.Map(2) − s_g^{1,2}] ∘ map_m ⟹ nMap(i) = N
  // G^{1,2}.depStates(2) = {x_l^1 | 2 ≤ l ≤ N}, X̂ = {x_N^1}
  // F^{1,2}.Map(2) = nMap, F^{1,2}.depStates(2) = {x_N^1}
  if N ≤ N ≤ N then
    xind = xMap(1, N) //nMap(row)=N
    if xind ∉ df^{1,1}dx, index(row, ·) then
      df^{1,2}dx.size(row) = df^{1,2}dx.size(row) + 1
      col = df^{1,2}dx.size(row)
      df^{1,2}dx.index(row, col) = xind
    else
      col = pos(xind in df^{1,2}dx.index(row, ·))
    end if
    rowg = row
```

```
    colg = pos(xind in dg^{1,2}dx.index(rowg,·))
    df^{1,2}dx(row,col) = df^{1,2}dx(row,col) + aux · dg^{1,2}dx(rowg,colg)
  end if
  // the other successor of F^{1,2} is X^2
  // the only map is map_m = map_1[f^{1,2},x^2](1) = 1
  // F^{1,2}.depStates(3)={x_1^2}, F^{1,2}.Map(3)=map_1[f^{1,2},x^2]
  if 1 ≤ 1 ≤ 1 then // because map_m(row) = 1 and X̂.indices=1
    xind=xMap(2,1) // because map_m(row) = 1
    if xind ∉ df^{1,2}dx.index(row,·) then
      df^{1,2}dx.size(row) = df^{1,2}dx.size(row) + 1
      col = df^{1,2}dx.size(row)
      df^{1,2}dx.index(row,col) = xind
    else
      col = pos(xind in df^{1,2}dx.index(row,·))
    end if
   df^{1,2}dx(row,col) = df^{1,2}dx(row,col) - 1/C //∂f_i^{1,2}/∂x_i^2
  end if
  // F^{1,2}.numDeps=3
end for
// end AddCode(F^{1,2},f^{1,2}), back to main algorithm
// call AddCode(F^2,1,f^{2,1}))
// the only successor of F^{2,1} is X^1
for row = 1 : 1
  // the only map is map_m = map_1[f^{2,1},x^1](1) = N
  // F^{2,1}.depStates(1)={x_N^1}, F^{2,1}.Map(1)=map_1[f^{2,1},x^1]
  if N ≤ N ≤ N then // because map_m(row) = N and X̂.indices=N
    xind=xMap(1,N) // because map_m(row) = N
    if xind ∉ df^{2,1}dx.index(row,·) then
      df^{2,1}dx.size(row) = df^{2,1}dx.size(row) + 1
      col = df^{2,1}dx.size(row)
      df^{2,1}dx.index(row,col) = xind
    else
      col = pos(xind in df^{2,1}dx.index(row,·))
    end if
    df^{2,1}dx(row,col) = df^{2,1}dx(row,col) + 1/L //∂f_i^{2,1}/∂x_N^1
  end if
  // F^{2,1}.numDeps=1
end for
// end AddCode(F^{2,1},f^{2,1}), back to main algorithm
// Copy partial derivatives into Jacobian matrix
r = 0
// i=1,j=1
for row = r+1 : r + N-1 //F^{1,1}.size = N-1
  for col = 1 : df^{1,1}dx.size(row-r)
    index(row,col) = df^{1,1}dx.index(row-r,col)
    J(row,col) = df^{1,1}dx(row-r,col)
  end for
  size(row) = df^{1,1}dx.size(row-r)
end for
r = r + N -1
// i=1,j=2
for row = r+1 : r + 1 //F^{1,2}.size = 1
  for col = 1 : df^{1,2}dx.size(row-r)
    index(row,col) = df^{1,2}dx.index(row-r,col)
    J(row,col) = df^{1,2}dx(row-r,col)
  end for
  size(row) = df^{1,2}dx.size(row-r)
```

```
end for
r = r + 1
// i=1,j=2
for row = r + 1 : r + 1 //F^{2,1}.size = 1
  for col = 1 : df^{2,1}dx.size(row − r)
    index(row, col) = df^{2,1}dx.index(row − r, col)
    J(row, col) = df^{2,1}dx(row − r, col)
  end for
  size(row) = df^{2,1}dx.size(row − r)
end for
```