Engineering with Computers (2022) 38:4423–4449
https://doi.org/10.1007/s00366-021-01558-6

ORIGINAL ARTICLE

IFOSMONDI Co-simulation Algorithm with Jacobian-Free Methods in PETSc

Yohan Eguillon[1,2] · Bruno Lacabanne[2] · Damien Tromeur-Dervout[1]

Received: 19 March 2021 / Accepted: 22 November 2021 / Published online: 8 April 2022
© The Author(s), under exclusive licence to Springer-Verlag London Ltd., part of Springer Nature 2022

Abstract
Co-simulation is a widely used solution to enable global simulation of a modular system via the composition of black-boxed simulators. Among co-simulation methods, the IFOSMONDI implicit iterative algorithm, previously introduced by the authors, enables us to solve the non-linear coupling function while keeping the smoothness of interfaces without introducing a delay. Moreover, it automatically adapts the size of the steps between data exchanges among the subsystems according to the difficulty of solving the coupling constraint. The latter was solved by a fixed-point algorithm, whereas this paper introduces the *Jacobian-Free Methods* version. Most implementations of Newton-like methods require a jacobian matrix which, except in the Zero-Order-Hold case, can be difficult to compute in the co-simulation context. As IFOSMONDI coupling algorithm uses Hermite interpolation for smoothness enhancement, we propose hereafter a new formulation of the non-linear coupling function including both the values and the time-derivatives of the coupling variables. This formulation is well designed for solving the coupling through jacobian-free Newton-type methods. Consequently, successive function evaluations consist in multiple simulations of the systems on a co-simulation time-step using rollback. The orchestrator-workers structure of the algorithm enables us to combine the PETSc framework on the orchestrator side for the non-linear Newton-type solvers with the parallel integrations of the systems on the workers' side thanks to MPI processes. Different non-linear methods will be compared to one another and to the original fixed-point implementation on a newly proposed 2-system academic test case with direct feedthrough on both sides. An industrial model will also be considered to investigate the performance of the method.

Keywords Co-simulation · Systems coupling · Coupling methods · Jacobian-free Newton · PETSc · Parallel integration · Strong coupling test case

1 Introduction

The use of co-simulation is increasing in the industry as it enables to connect and simulate systems with given interfaces (input and output variables) without disclosing the expertise inside. Hence, modellers can provide system architects with virtual systems as black-boxes since the systems are able to interact through their interfaces. Among these interactions, the minimal requirements are quite simple: a system should at least be able to read the inputs given by the other systems, to simulate its physics inside (most of the time thanks to an embedded solver), and to provide outputs of the simulation to the other systems.

Besides its black-box aspect protecting the know-how, co-simulation also enables physic-based decomposition (one system can represent the hydraulic part of a modular model, another the mechanical part, a third one the electrical part, and so on) and/or dynamics-based decomposition (some systems isolate the stiff state variables so that they do not constraint all the other states anymore during the simulation). In other words, the co-simulation opens many doors thanks to the modular aspect of the models handled.

Supported by organization Siemens Industry Software.

✉ Yohan Eguillon
yohan.eguillon@univ-lyon1.fr; yohan.eguillon@siemens.com

Bruno Lacabanne
bruno.lacabanne@siemens.com

Damien Tromeur-Dervout
damien.tromeur-dervout@univ-lyon1.fr

[1] Institut Camille Jordan, Université de Lyon, UMR5208 CNRS-U.Lyon1, Villeurbanne, France

[2] Siemens Industry Software, Roanne, France

The co-simulation field of research nowadays focuses on the numerical methods and algorithms that can be used to process simulations of such modular models. From the simplest implementations (non-iterative Jacobi) to very advanced algorithms [4, 7, 11–13, 15], co-simulation methods have been developed in different fields, showing that the underlying problems to be tackled are not straightforward. Some arising problems could clearly be identified since the moment it has become a center of interest for researchers, such as the delay between the given inputs and the retrieved outputs of a system (corresponding to the so-called "co-simulation step" or "macro-step"), the instabilities that might occur as a consequence of this delay [16], the discontinuities produced at each communication [5], the error estimation (and the use of it to adapt the macro-step size) [13], the techniques to solve the so-called "constraint function" corresponding to the interface of the systems [9, 14], and so on. Moreover, performance issues usually arise when co-simulation codes are implemented in practice, for instance: idling systems (in Gauss-Seidel-like methods systems are simulated sequentially, one at a time, and in Jacobi-like methods the time taken by a co-simulation step is the one of the slowest system due to synchronization points where faster systems have to wait for slower ones). Many of these problems have been addressed in papers, either proposing an analysis, a method to solve them, or both.

In our previous paper [6], an iterative method that satisfies the interfaces' consistency while avoiding discontinuities at each macro-step was proposed and compared to well-established methods (non-iterative Jacobi, zero-order hold iterative co-simulation [9], and non-iterative algorithm enhancing variables' smoothness [5]). This algorithm was based on a fixed-point iterative method. Its evolution, presented in this paper, is based on iterative methods that normally require jacobian matrix computation, yet we use their jacobian-free version. The name of this method is IFOSMONDI-JFM, standing for *Iterative and Flexible Order, SMOoth and Non-Delayed Interfaces, based on Jacobian-Free Methods*. The enhancements it brings to the classical IFOSMONDI method enable to solve cases that could not be solved by this previous version. The integration of an easily modulable jacobian-free method to solve the constraint function will be presented. The software integration, in particular, was made possible thanks to the PETSc framework, a library that provides modulable numerical algorithms. The interfacing between PETSc and the co-simulation framework dealing with the systems, interfaces and polynomial representations will be detailed.

2 Formalism and notations

2.1 A word on the JFM accronym

In the whole paper, the JFM abbreviation will denote jacobian-free versions of iterative methods that are designed to bring a given function (the so-called *callback*) to zero and that normally require the computation of the jacobian matrix of the callback function. In particular, a fixed-point method does not meet these criteria: it is not a JFM, contrary to matrix-free versions of the Newton method, the Anderson method [1] or the non-linear GMRES method [10].

2.2 General notations

The set $M_{a,b}(A)$ will represent the set of matrices of a rows and b columns with its coefficients in the set A.

In this paper, we will focus on explicit systems. In other words, we will consider that every system in the co-simulation is a dynamical system corresponding to an ODE (Ordinary Differential Equation). The time-domain of the ODEs considered will be written $[t^{\text{init}}, t^{\text{end}}[$, and the variable t will denote the time.

Let's consider $n_{sys} \in \mathbb{N}^*$ systems are involved: we will use the index $k \in [\![1, n_{sys}]\!]$ to denote the kth system, and $n_{st,k}$, $n_{in,k}$, and $n_{out,k}$ will respectively denote the number of state variables, the number of inputs, and the number of outputs of system k.

The time-dependant vectors of states, inputs and outputs of system k will, respectively, be written $x_k \in L([t^{\text{init}}, t^{\text{end}}[, \mathbb{R}^{n_{st,k}})$, $u_k \in L([t^{\text{init}}, t^{\text{end}}[, \mathbb{R}^{n_{in,k}})$, and $y_k \in L([t^{\text{init}}, t^{\text{end}}[, \mathbb{R}^{n_{out,k}})$ where $L(A, B)$ denotes the set of functions of domain A and co-domain B. We can write the ODE form of the system k:

$$\begin{cases} \dot{x}_k(t) &= f_k(t, x_k(t), u_k(t)) \\ y_k(t) &= g_k(t, x_k(t), u_k(t)) \end{cases} \tag{1}$$

Please note that co-simulation is mainly interesting on 0D systems. Indeed, CFD systems for instance can be split in term of physics, generating systems coupled on every point in space. This would generate a very high number of interfaces $n_{in,k}$ and $n_{out,k}$ for all k in $[\![1, n_{sys}]\!]$. Although co-simulation can work on such cases, we will focus on 0D systems (such as the test-cases presented in Sect. 5) as co-simulation becomes relevant when the stiffness of the systems are local on each of them, and where the interface variables (inputs and outputs) are smooth and relatively few.

Let $n_{in,tot}$ and $n_{out,tot}$ respectively be the total amount of inputs $\sum_{k=1}^{n_{sys}} n_{in,k}$ and the total amount of outputs $\sum_{k=1}^{n_{sys}} n_{out,k}$.

The total input and the total output vectors are simply concatenations of input and output vectors of every system. They will be denoted by underlined vectors. The underline will denote a quantity *"upon every subsystem"*.

$$\begin{aligned} \underline{u}(t) &= (u_1(t)^T, \dots, u_{n_{sys}}(t)^T)^T &\in L([t^{\text{init}}, t^{\text{end}}[, \mathbb{R}^{n_{in,tot}}) \\ \underline{y}(t) &= (y_1(t)^T, \dots, y_{n_{sys}}(t)^T)^T &\in L([t^{\text{init}}, t^{\text{end}}[, \mathbb{R}^{n_{out,tot}}) \end{aligned} \tag{2}$$

To illustrate the notations introduced above, an example is given further in this paper, in Fig. 3.

Finally, a tilde symbol $\tilde{}$ will be added to a functional quantity to represent an element of its co-domain. *exempli gratia*, $\underline{y} \in L([t^{[N]}, t^{[N+1]}[, \mathbb{R})$, so we can use $\tilde{\underline{y}}$ to represent an element of $\mathbb{R}^{n_{out,tot}}$.

2.3 Extractors and rearrangement

To easily switch from global to local inputs, extractors are defined. For $k \in [\![1, n_{sys}]\!]$, the extractor E_k^u is the matrix defined by (3).

$$E_k^u = \Big(\underbrace{0}_{n_{in,k} \times \sum_{l=1}^{k-1} n_{in,l}} \;\Big|\; \underbrace{\left(I_{n_{in,k}}\right)}_{n_{in,k} \times n_{in,k}} \;\Big|\; \underbrace{0}_{n_{in,k} \times \sum_{l=k+1}^{n_{sys}} n_{in,l}} \Big) \tag{3}$$

where $\forall n \in \mathbb{N},\ I_n$ denotes the identity matrix of size n by n.

The extractors enable to extract the inputs of a given system from the global inputs vector with a relation of the form $\tilde{u}_k = E_k^u \tilde{\underline{u}}$. We have: $\forall k \in [\![1, n_{sys}]\!],\ E_k^u \in M_{n_{in,k}, n_{in,tot}}(\{0, 1\})$.

A rearrangement operator will also be needed to handle concatenations of outputs and output derivatives. For this purpose, we will use the rearrangement matrix $R^{\underline{y}} \in M_{n_{out,tot}, n_{out,tot}}(\{0, 1\})$ defined blockwise in (4).

$$R^{\underline{y}} = \left(R^{\underline{y}}_{K,L}\right)_{\substack{K \in [\![1,\ 2\, n_{sys}]\!] \\ L \in [\![1,\ 2\, n_{sys}]\!]}}$$

where

$$R^{\underline{y}}_{K,L} = \begin{cases} I_{n_{out,K}} & \text{if } K \leqslant n_{sys} \text{ and } L = 2K-1 \\ I_{n_{out,K-n_{sys}}} & \text{if } K > n_{sys} \text{ and } L = 2(K - n_{sys}) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The $R^{\underline{y}}$ operator makes it possible to rearrange the outputs and output derivatives with a relation of the form (5).

$$\begin{pmatrix} \tilde{\underline{y}} \\ \text{—} \\ \tilde{\dot{\underline{y}}} \end{pmatrix} = \left(\begin{array}{c} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_{n_{sys}} \\ \hline \tilde{\dot{y}}_1 \\ \tilde{\dot{y}}_2 \\ \vdots \\ \tilde{\dot{y}}_{n_{sys}} \end{array}\right) = \underbrace{\left(\begin{array}{ccccccc} I_{n_{out,1}} & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & I_{n_{out,2}} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & I_{n_{out,n_{sys}}} & 0 \\ \hline 0 & I_{n_{out,1}} & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & I_{n_{out,2}} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & I_{n_{out,n_{sys}}} \end{array}\right)}_{R^{\underline{y}}} \begin{pmatrix} \tilde{y}_1 \\ \tilde{\dot{y}}_1 \\ \tilde{y}_2 \\ \tilde{\dot{y}}_2 \\ \vdots \\ \tilde{y}_{n_{sys}} \\ \tilde{\dot{y}}_{n_{sys}} \end{pmatrix} \tag{5}$$

As explained in Sect. 2.2, a reduced number of inputs and outputs is advised. On the extractors and rearrangement operators, a high number of interface variables will produce large matrices. However, the operations implying such matrices will not be strongly impacted as they are not explicitely constructed in practice. Indeed, these operators help to define the mathematical formalism (namely the callback function, further introduced in Sect. 3.2) yet in practice the application of the extractor operators can be done while communicating with a simple call to `MPI_Scatterv` function, and the application of the rearrangement operator can be done while communicating with a simple call to `MPI_Gatherv` function (such workflow will be presented further in this paper and illustrated in Fig. 7). Lastly, in the implementation, the E_k^u (for all k in $[\![1, n_{sys}]\!]$) and $R^{\underline{y}}$ matrices will never be assembled.

2.4 Time discretization

In the context of co-simulation, the g_k and f_k functions in (1) are usually not available directly. Thus, several co-simulation steps, the so-called "macro-steps", are made between t^{init} and t^{end}. Let's introduce the notations of the discrete version of the quantities introduced in Sect. 2.2.

A macro-step will be defined by its starting and ending times, respectively denoted as $[t^{[N]}, t^{[N+1]}]$ for the Nth macro-step. The subscript $^{[N]}$ will be written with square brackets to

avoid confusion with power exponents (*exempli gratia*: t^2). The macro-steps define a partition of the time-domain, as described in (6) and Fig. 1.

$$\begin{cases} [t^{\text{init}}, t^{\text{end}}[& = \bigcup_{N=0}^{N_{\max}-1} [t^{[N]}, t^{[N+1]}[\\ t^{[0]} & = t^{\text{init}} \\ t^{[N_{\max}]} & = t^{\text{end}} \\ \forall N \in [\![0, N_{\max}-1]\!],\ t^{[N+1]} > t^{[N]} \end{cases} \tag{6}$$

Let $\delta t^{[N]}$ denote the size of the Nth macro-step:

$$\begin{cases} \forall N \in [\![0, N_{\max}-1]\!],\ \delta t^{[N]} = t^{[N+1]} - t^{[N]} > 0 \\ \sum_{N=0}^{N_{\max}-1} \delta t^{[N]} = t^{\text{end}} - t^{\text{init}} \end{cases} \tag{7}$$

Let $\mathbb{T}$ denote the set of possible macro-steps.

$$\mathbb{T} \stackrel{\Delta}{=} \{[a, b[\mid t^{\text{init}} \leqslant a < b \leqslant t^{\text{end}}\} \tag{8}$$

An element of this set is a macro-step: for instance $\tau \in \mathbb{T}$ with $\tau = [t^{[N]}, t^{[N+1]}[$.

On a given macro-step $[t^{[N]}, t^{[N+1]}[$, $N \in [0, N_{\max}]$, for all systems, the restrictions of the piecewise equivalents of u_k and y_k will be denoted by $u_k^{[N]}$ and $y_k^{[N]}$ respectively. In case several iterations are made on the same step, we will refer to the functions by a left superscript index m. Finally, we will denote the coordinate of these vectors with an extra subscript index.

$$\begin{aligned} &\forall k \in [\![1, n_{sys}]\!], \forall N \in [\![0, N_{\max}]\!],\ \forall m \in [0, m_{\max}(N)], \\ &\forall j \in [\![1, n_{in,k}]\!], {}^{[m]}u_{k,j}^{[N]} \in L([t^{[N]}, t^{[N+1]}[, \mathbb{R}) \\ &\forall i \in [\![1, n_{out,k}]\!], {}^{[m]}y_{k,i}^{[N]} \in L([t^{[N]}, t^{[N+1]}[, \mathbb{R}) \end{aligned} \tag{9}$$

In (9), $m_{\max}(N)$ denotes the number of iterations (minus one) done on the Nth macro-step. $m_{\max}(N)$ across N can be plotted in order to see where the method needs to proceed more or less iterations.

All derived notations introduced in this subsection can also be applied to the total input and output vectors.

$$\begin{aligned} &\forall N \in [\![0, N_{\max}]\!],\ \forall m \in [\![0, m_{\max}(N)]\!], \\ &\forall \bar{j} \in n_{in,tot}, {}^{[m]}\underline{u}_{\bar{j}}^{[N]} \in L([t^{[N]}, t^{[N+1]}[, \mathbb{R}) \\ &\forall \bar{\imath} \in n_{out,tot}, {}^{[m]}\underline{y}_{\bar{\imath}}^{[N]} \in L([t^{[N]}, t^{[N+1]}[, \mathbb{R}) \end{aligned} \tag{10}$$

Indices $\bar{\imath}$ and $\bar{j}$ in (10) will be called *global indices* in opposition to the *local indices* i and j in (9).

2.5 Step function

Let S_k, $k \in [\![1, n_{sys}]\!]$ be the *ideal step function* of the kth system, that is to say the function which takes the system to its future state one macro-step forward.

$$S_k : \begin{cases} \mathbb{T} \times L([t^{\text{init}}, t^{\text{end}}[, \mathbb{R}^{n_{in,k}}) \times \mathbb{R}^{n_{st,k}} & \to \mathbb{R}^{n_{out,k}} \times \mathbb{R}^{n_{st,k}} \\ (\tau,\ u_k,\ \tilde{x}) & \mapsto S_k(\tau,\ u_k,\ \tilde{x}) \end{cases} \tag{11}$$

In practice, the state vector $\tilde{x}$ will not be explicited. Indeed, it will be embedded inside of the system k and successive calls will either be done:

- with τ beginning where the τ at the previous call of S_k ended (moving on),
- with τ beginning where the τ at the previous call of S_k started (step replay),
- with τ of the shape $[t^{\text{init}}, t^{[1]}[$ with $t^{[1]} \in]t^{\text{init}}, t^{\text{end}}]$ (first step).

Moreover, the u_k argument only needs to be defined on domain τ (not necessarily on $[t^{\text{init}}, t^{\text{end}}[$). Thus, S_k will not be considered in the method, but the $\hat{S}_k$ function (*practical step function)* defined hereafter will be considered instead. Despite $\hat{S}_k$ is not properly mathematically defined (the domain depends on the value of one of the arguments: τ and some quantities are hidden: the states), it does not lead to any problem, considering the hypotheses above.

$$\hat{S}_k : \begin{cases} \mathbb{T} \times L(\tau, \mathbb{R}^{n_{in,k}}) & \mapsto \mathbb{R}^{n_{out,k}} \\ (\tau, u_k) & \mapsto \hat{S}_k(\tau, u_k) \end{cases}$$

$$\text{satisfying} \tag{12}$$

$$\hat{S}_k([t^{[N]}, t^{[N+1]}[,\ {}^{[m]}u_k^{[N]}) = {}^{[m]}y_k^{[N]}(t^{[N+1]})$$

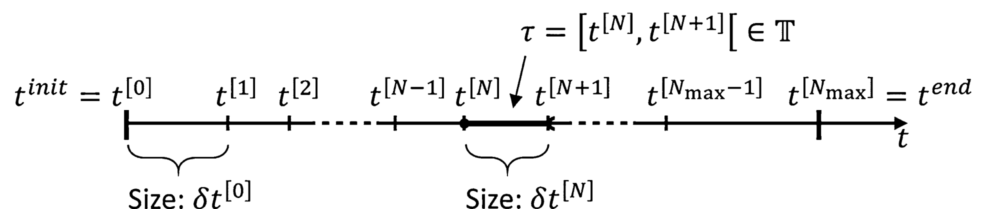

Fig. 1 Partition of the time domain in macro-steps

The $\hat{S}_k$ function is the one available in practice, namely in the FMI (Functional Mock-up Interface) standard.

2.6 Extended step function

The values of the output variables might not be sufficient for every co-simulation scheme. It is namely the case for both classical IFOSMONDI and IFOSMONDI-JFM. Indeed, the time-derivatives of the outputs are also needed.

Let $\hat{\hat{S}}_k$ be the extension of $\hat{S}_k$ returning both the output values and derivatives.

$$\hat{\hat{S}}_k : \begin{cases} \mathbb{T} \times L(\tau, \mathbb{R}^{n_{in,k}}) & \mapsto \mathbb{R}^{n_{out,k}} \times \mathbb{R}^{n_{out,k}} \\ (\tau, u_k) & \mapsto \hat{\hat{S}}_k(\tau, u_k) \end{cases}$$

satisfying

$$\hat{\hat{S}}_k([t^{[N]}, t^{[N+1]}[, {}^{[m]}u_k^{[N]}) = \begin{pmatrix} {}^{[m]}y_k^{[N]}(t^{[N+1]}) \\ \dfrac{d\ {}^{[m]}y_k^{[N]}}{dt}(t^{[N+1]}) \end{pmatrix} \tag{13}$$

To evaluate $\hat{\hat{S}}_k$, system k is integrated over the time-domain τ (first argument) with inputs given by the second argument: vectorial function (of dimension $n_{in,k}$) of the time, and the values and derivatives of the outputs (vectorial function of the time, of dimension $n_{out,k}$) are returned, evaluated at the time corresponding to the end of the first argument ($\sup(\tau)$). Figure 2 presents an example of this workflow.

2.7 Connections

The connections between the systems will be denoted by a matrix filled with zeros and ones, with $n_{out,tot}$ rows and $n_{in,tot}$ columns denoted by $\boldsymbol{\Phi}$. Please note that if each output is connected to exactly one input, $\boldsymbol{\Phi}$ is a square matrix. Moreover, it is a permutation matrix. Otherwise, if an output is connected to several inputs, more than one 1 appears at the corresponding row of $\boldsymbol{\Phi}$. Without loss of generality, let's consider that there can neither be more nor less than one 1 on each column of $\boldsymbol{\Phi}$ considering that an input can neither be connected to none nor several outputs. Indeed, a system with an input connected to nothing is not possible (a value has to be given), and a connection of several outputs in the same input can always be decomposed regarding a relation (sum, difference, ...) so that this situation is similar to distinct inputs connected to a single output each, with these inputs are combined (added, substracted, ...) inside of the system considered.

$$\forall \bar{\imath} \in n_{out,tot}, \ \forall \bar{\jmath} \in n_{in,tot}, \quad \boldsymbol{\Phi}_{\bar{\imath},\bar{\jmath}} = \begin{cases} 1 & \text{if output } \bar{\imath} \text{ is connected to input } \bar{\jmath} \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

An example of a connection matrix is presented in Fig. 3.

The *dispatching* will denote the stage where the inputs are generated from their connected inputs, using the connections represented by $\boldsymbol{\Phi}$.

$$\underline{\tilde{u}} = \boldsymbol{\Phi}^T \underline{\tilde{y}} \tag{15}$$

Analogously to the extractor and rearrangement operators introduced in Sect. 2.3, the $\boldsymbol{\Phi}$ matrix does not need to be explicitly constructed in practice. Indeed, the implementation only needs to know the connections to proceed with the dispatching (15).

The *coupling function* (16) will denote the absolute difference between corresponding connected variables in a total input vector and a total output vector. In other words, it represents the absolute error between a total input vector and the dispatching of a total output vector. The λ subscript does not correspond to a quantity, it is a simple notation inherited from a "Lagrange multipliers" approach of system coupling [14].

$$g_\lambda : \begin{cases} \mathbb{R}^{n_{in,tot}} \times \mathbb{R}^{n_{out,tot}} & \rightarrow \mathbb{R}^{n_{in,tot}} \\ (\underline{\tilde{u}}, \underline{\tilde{y}}) & \mapsto |\underline{\tilde{u}} - \boldsymbol{\Phi}^T \underline{\tilde{y}}| \end{cases} \tag{16}$$

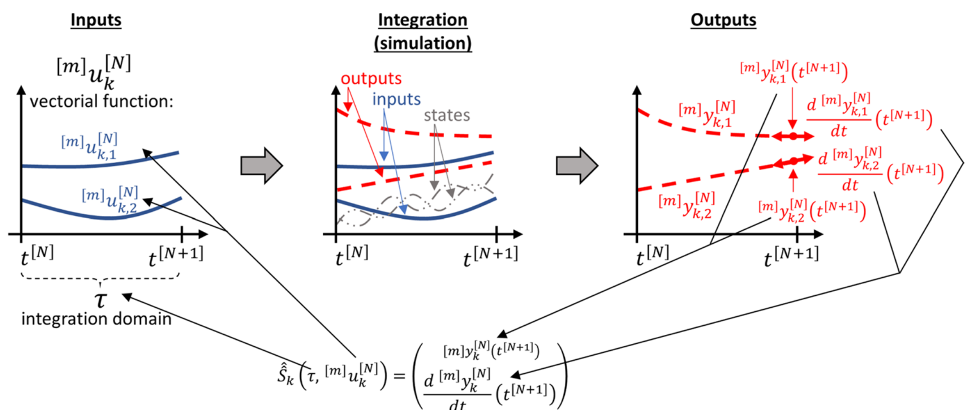

Fig. 2 Extended step function's workflow visualization on an example where system k has 2 inputs ($n_{in,k} = 2$), 2 states, and 2 outputs ($n_{out,k} = 2$)

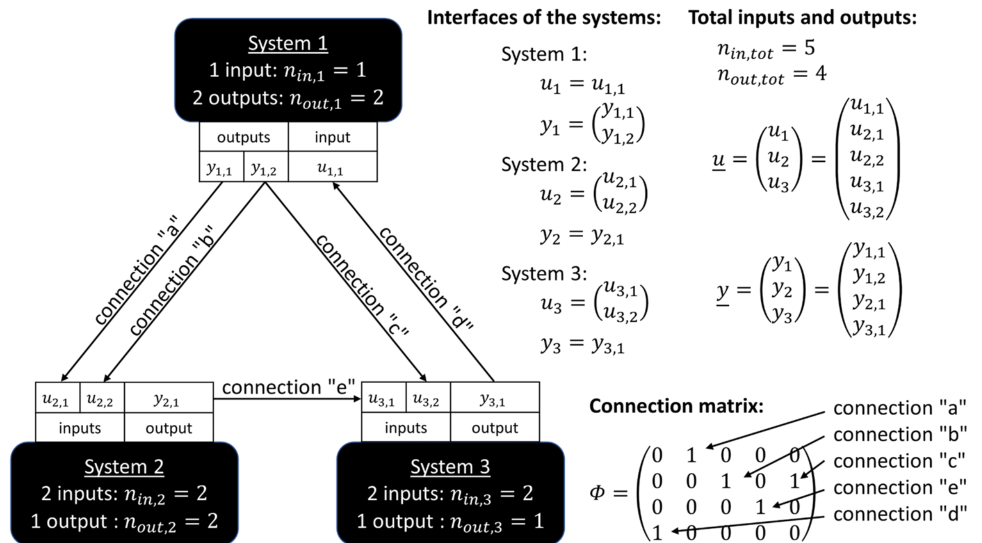

Fig. 3 Example of a 3-system co-simulation model with its interfaces and its Φ matrix

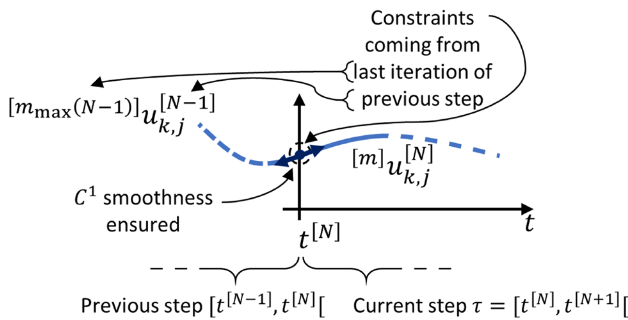

Fig. 4 C^1 smoothness constraints at the left of τ for j^{th} input of system k

The *coupling condition* (17) is the situation where every output of the total output vector corresponds to its connected input in the total input vector.

$$g_\lambda(\underline{\tilde{u}}, \underline{\tilde{y}}) = 0_{\mathbb{R}^{n_{in,tot}}} \tag{17}$$

3 IFOSMONDI-JFM method

3.1 Modified extended step function

As in classical IFOSMONDI [6], the IFOSMONDI-JFM method preserves the C^1 smoothness of the interface variables at the communication times $(t^{[N]})_{N \in [\![1, N_{\max}-1]\!]}$. Thus, when a time $t^{[N]}$ has been reached, the input functions for every system will all satisfy the property (18) illustrated in Fig. 4.

$$\forall k \in [\![1, n_{sys}]\!],\ \forall m \in [\![0, m_{\max}(N)]\!],$$
$$\begin{cases} {}^{[m]}u_k^{[N]}(t^{[N]}) &= {}^{[m_{\max}(N-1)]}u_k^{[N-1]}(t^{[N]}) \\ \dfrac{d\ {}^{[m]}u_k^{[N]}}{dt}(t^{[N]}) &= \dfrac{d\ {}^{[m_{\max}(N-1)]}u_k^{[N-1]}}{dt}(t^{[N]}) \end{cases} \tag{18}$$

The IFOSMONDI-JFM method also represents the inputs as 3rd order polynomial (maximum) to satisfy the smoothness condition (18) and to respect imposed values and derivatives at $t^{[N+1]}$ for every macro-step.

Knowing these constraints, it is possible to write a specification of the practical step function $\hat{\hat{S}}_k$ in the IFOSMONDI-JFM case (also applicable in the classical IFOSMONDI method):

$$\zeta_k : \begin{cases} \mathbb{T} \times \mathbb{R}^{n_{in,k}} \times \mathbb{R}^{n_{in,k}} & \mapsto \mathbb{R}^{n_{out,k}} \times \mathbb{R}^{n_{out,k}} \\ (\tau, \tilde{u}_k, \tilde{\dot{u}}_k) & \mapsto \zeta_k(\tau, \tilde{u}_k, \tilde{\dot{u}}_k) \end{cases} \tag{19}$$

where the three cases discussed in Sect. 2.5 have to be considered.

Once each of these cases has been detailed, Figs. 5 and 6 will show the succession of such cases.

3.1.1 Case 1: Moving on

In this case, the last call to ζ_k was done with a $\tau \in \mathbb{T}$ ending at current $t^{[N]}$. In other words, the system k “reached” time $t^{[N]}$. The inputs were, at this last call: ${}^{[m_{\max}(N-1)]}u_k^{[N-1]}$.

To reproduce a behavior analog to that of the classical IFOSMONDI method, the inputs ${}^{[0]}u_k^{[N]}$ will be defined as the 2nd order polynomial (or less) satisfying the three following constraints:

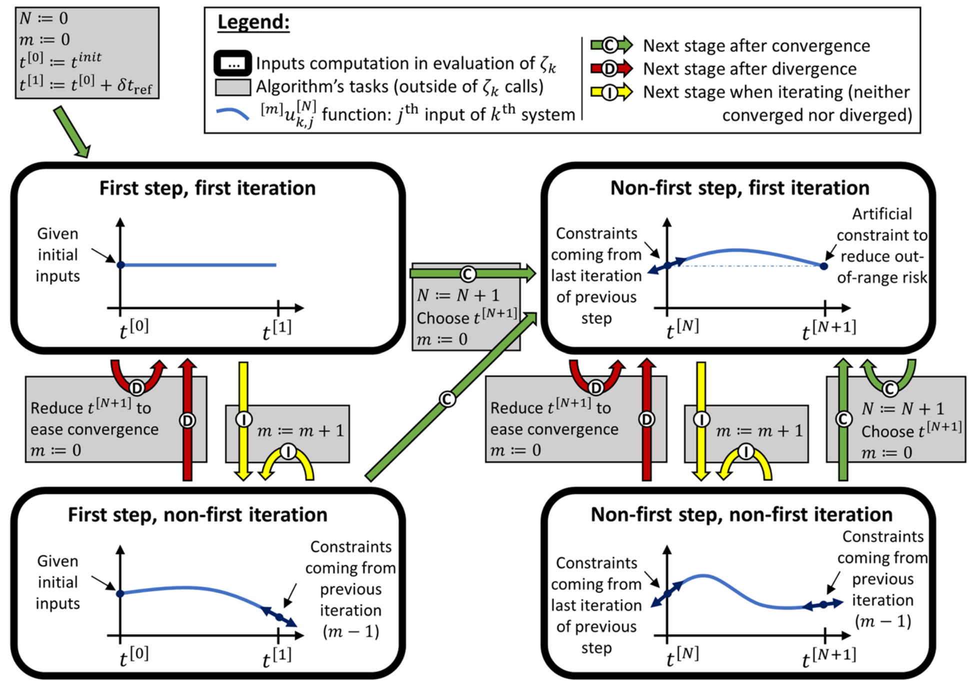

Fig. 5 Workflow of the calibration of the inputs, visualization on a single given jth input of a given system k, and algorithm's tasks in transitions between cases. This figure does not represent the whole method: it only focuses on the inputs calibration

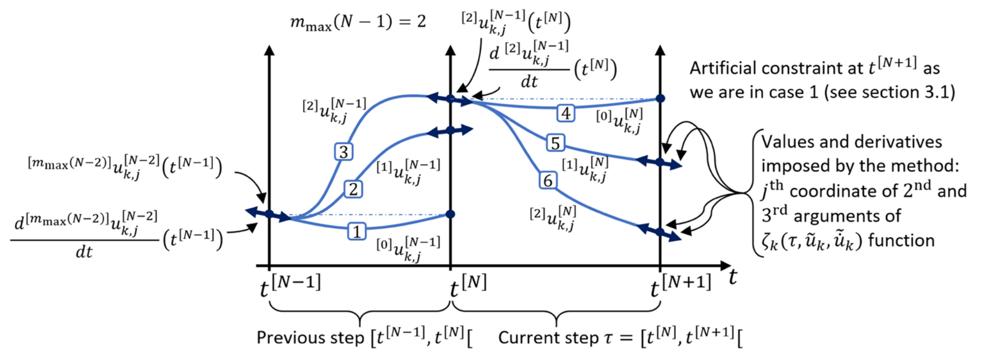

Fig. 6 Focus on a single input j of system k: on the co-simulation step τ, it can be seen that the constraints at the beginning of the steps come from the last iteration of the previous co-simulation step, and the constraints at the end of the steps come from the method, or are artificial (for the first iteration)

$$
\begin{aligned}
{}^{[0]}u_k^{[N]}(t^{[N]}) &= {}^{[m_{\max}(N-1)]}u_k^{[N-1]}(t^{[N]}) \\
\frac{d\ {}^{[0]}u_k^{[N]}}{dt}(t^{[N]}) &= \frac{d\ {}^{[m_{\max}(N-1)]}u_k^{[N-1]}}{dt}(t^{[N]}) \\
{}^{[0]}u_k^{[N]}(t^{[N+1]}) &= {}^{[m_{\max}(N-1)]}u_k^{[N-1]}(t^{[N]})
\end{aligned}
\quad (20)
$$

The two first constraints guarantee the smoothness property (18), and the third one minimizes the risk of out-of-range values (as in the classical IFOSMONDI method).

In this case, ζ_k in (19) is defined by the specification (21).

$$
\zeta_k([t^{[N]}, t^{[N+1]}[, \cdot, \cdot) = \hat{\hat{S}}_k([t^{[N]}, t^{[N+1]}[, \underbrace{{}^{[0]}u_k^{[N]}}_{\text{computed with (20)}}) \quad (21)
$$

2nd and 3rd arguments of ζ_k are unused.

3.1.2 Case 2: Step replay

In this case, the last call to ζ_k was done with a $\tau \in \mathbb{T}$ starting at current $t^{[N]}$. In other words, the system did not manage to reach the ending time of the previous τ (either because the method did not converge, or because the step has been rejected, or another reason).

Two particular subcases have to be considered here: either the step we are computing is following the previous one in the iterative method detailed after this section, or the previous iteration has been rejected and we are trying to re-integrate the step starting from τ with a smaller size $\delta t^{[N]}$.

3.1.2.1 Subcase 2.1: Following a previous classical step

In this subcase, the last call of ζ_k was not only done with the same starting time, but also with the same step ending time $t^{[N+1]}$. The inputs were, at this last call: ${}^{[m-1]}u_k^{[N]}$ with $m \geqslant 1$, and satisfied the two conditions at $t^{[N]}$ of (21).

The jacobian-free iterative method will ask for given input values $\tilde{u}_k$ and time-derivatives $\tilde{\dot{u}}_k$ that will be used as constraints at $t^{[N+1]}$, thus ${}^{[m]}u_k^{[N]}$ will be defined as the 3rd order polynomial (or less) satisfying the four constraints depicted in (22).

$$\begin{array}{lllll} {}^{[m]}u_k^{[N]}(t^{[N]}) & = & {}^{[m_{\max}(N-1)]}u_k^{[N-1]}(t^{[N]}) & = & {}^{[m-1]}u_k^{[N]}(t^{[N]}) \\ \frac{d\,{}^{[m]}u_k^{[N]}}{dt}(t^{[N]}) & = & \frac{d\,{}^{[m_{\max}(N-1)]}u_k^{[N-1]}}{dt}(t^{[N]}) & = & \frac{d\,{}^{[m-1]}u_k^{[N]}}{dt}(t^{[N]}) \\ {}^{[m]}u_k^{[N]}(t^{[N+1]}) & = & \tilde{u}_k & & \\ \frac{d\,{}^{[m]}u_k^{[N]}}{dt}(t^{[N+1]}) & = & \tilde{\dot{u}}_k & & \end{array} \tag{22}$$

The two firsts constraints ensure the (18) smoothness property, and the third and fourth one will enable the iterative method to find the best values and derivatives to satisfy the coupling condition.

In this subcase, ζ_k in (19) is defined by the specification (23).

$$\zeta_k([t^{[N]}, t^{[N+1]}[, \tilde{u}_k, \tilde{\dot{u}}_k) = \hat{\hat{S}}_k([t^{[N]}, t^{[N+1]}[, \underbrace{{}^{[m]}u_k^{[N]}}_{\text{computed with (22)}}) \tag{23}$$

3.1.2.2 Subcase 2.2: Re-integrate a step starting from $t^{[N]}$ but with different $\delta t^{[N]}$ than at the previous call of ζ_k

In this subcase, current $t^{[N+1]}$ is different from $\sup(\tau)$ with τ being the one used at the last call of ζ_k.

As it shows that a step rejection just occurred, we will simply do the same than in case 1, as if we were moving on from $t^{[N]}$. In other words, all calls to ζ_k with τ starting at $t^{[N]}$ are "forgotten".

Please note that ${}^{[m_{\max}(N-1)]}u_k^{[N-1]}(t^{[N]})$ and $\frac{d\,{}^{[m_{\max}(N-1)]}u_k^{[N-1]}}{dt}(t^{[N]})$ can be retrieved using the values and derivatives constraints at $t^{[N]}$ of the inputs at the last call of ζ_k thanks to the smoothness constraint (18).

3.1.3 Case 3: First step

In this particular case, we will do the same as in the other cases, except that we would not impose any constraint for the time-derivative at t^{init}. That is to say:

- at the first call of ζ_k, we have $N = m = 0$, we will only impose ${}^{[0]}u_k^{[0]}(t^{\text{init}}) = {}^{[0]}u_k^{[0]}(t^{[1]}) = u_k^{\text{init}}$ to have a zero order polynomial satisfying the initial conditions u_k^{init} (supposed given),
- at the other calls, case 2 will be used without considering the constraints for the derivatives at t^{init} (this will lower the polynomial's degrees). For (22), the first condition becomes ${}^{[m]}u_k^{[N]}(t^{\text{init}}) = u_k^{\text{init}}$, the second one vanishes, and the third ans fourth ones remain unchanged. For the subcase 2.2, it can be considered that ${}^{[m_{\max}(-1)]}u_k^{[-1]}(t^{\text{init}}) = u_k^{\text{init}}$, and $\frac{d\,{}^{[m_{\max}(-1)]}u_k^{[-1]}}{dt}(t^{\text{init}})$ will not be needed as it is a time-derivative in t^{init}.

Finally, we have ζ_k defined in every case, wrapping both the computation of the polynomial inputs and the integration done with $\hat{\hat{S}}_k$.

The workflow consisting in the succession of the cases detailed above can be visualized in Fig. 5. An example on a given single input of a given single system is presented in Fig. 6 on 2 successive co-simulation steps. Squared number 1 to 6 denote the order of the successive input computations.

Until here, the polynomial inputs computation stage during an evaluation of ζ_k for $k \in [\![1, n_{sys}]\!]$ has been detailed among all the possible cases. However, the constraints at the end of the co-simulation steps have been described as "coming from the method". Indeed, the JFM will decide of the constraints to use as they will exactly be the variables of the function to zero (the aforementioned callback function, see Sect. 3.2).

3.2 Iterative method's callback function

The aim is to solve the co-simulation problem by using a jacobian-free version of an iterative method that usually requires a jacobian computation (see Sect. 2.1). Modern matrix-free versions of such algorithms make it possible to avoid perturbating the systems and re-integrating them for every input, as done in [14], to compute a finite-differences jacobian matrix. This saves a lot of integrations over each macro-step and saves time.

Nevertheless, on every considered macro-step τ, a function to be brought to zero has to be defined. This so-called *JFM's callback* (standing for *Jacobian-Free Method's callback*) presented hereafter will be denoted by γ_τ. In zero-order hold co-simulation, this function if often $\underline{\tilde{u}} - \Phi^T \underline{\tilde{y}}$ (or equivalent) where $\underline{\tilde{y}}$ are the output at $t^{[N+1]}$ generated by constant inputs $\underline{\tilde{u}}$ over $[t^{[N]}, t^{[N+1]}[$.

In IFOSMONDI-JFM, we will only enable to change the inputs at $t^{[N+1]}$, the smoothness condition at $t^{[N]}$ guaranteeing that the coupling condition (17) remains satisfied at $t^{[N]}$ if it was satisfied before moving on to the step $[t^{[N]}, t^{[N+1]}[$. The time-derivatives will also be considered to maintain C^1 smoothness, so the coupling condition (17) will also be applied to these time-derivatives.

Finally, the formulation of the JFM's callback for IFOSMONDI-JFM is given in (24).

$$\gamma_\tau : \begin{cases} \mathbb{R}^{n_{in,tot}} \times \mathbb{R}^{n_{in,tot}} & \to \mathbb{R}^{n_{in,tot}} \times \mathbb{R}^{n_{in,tot}} \\ \begin{pmatrix} \underline{\tilde{u}} \\ \underline{\tilde{\tilde{u}}} \end{pmatrix} & \mapsto \begin{pmatrix} \underline{\tilde{u}} \\ \underline{\tilde{\tilde{u}}} \end{pmatrix} - \begin{pmatrix} \Phi^T & 0 \\ 0 & \Phi^T \end{pmatrix} R^{\underline{y}} \begin{pmatrix} \zeta_1\left(\tau, E_1^u \underline{\tilde{u}}, E_1^u \underline{\tilde{\tilde{u}}}\right) \\ \vdots \\ \zeta_{n_{sys}}\left(\tau, E_{n_{sys}}^u \underline{\tilde{u}}, E_{n_{sys}}^u \underline{\tilde{\tilde{u}}}\right) \end{pmatrix} \end{cases} \tag{24}$$

3.2.1 Link with the fixed-point implementation

The formulation (24) can be used to represent the expression of the fixed-point Ψ_τ function. The latter has been introduced in classical IFOSMONDI algorithm [6] where a fixed-point method was used instead of a JFM one.

We can now rewrite a proper expression of Ψ_τ including the time-derivatives.

$$\Psi_\tau : \begin{cases} \mathbb{R}^{n_{in,tot}} \times \mathbb{R}^{n_{in,tot}} & \to \mathbb{R}^{n_{in,tot}} \times \mathbb{R}^{n_{in,tot}} \\ \begin{pmatrix} \underline{\tilde{u}} \\ \underline{\tilde{\tilde{u}}} \end{pmatrix} & \mapsto \begin{pmatrix} \underline{\tilde{u}} \\ \underline{\tilde{\tilde{u}}} \end{pmatrix} - \gamma_\tau\left(\begin{pmatrix} \underline{\tilde{u}} \\ \underline{\tilde{\tilde{u}}} \end{pmatrix}\right) \\ & = \begin{pmatrix} \Phi^T & 0 \\ 0 & \Phi^T \end{pmatrix} R^{\underline{y}} \begin{pmatrix} \zeta_1\left(\tau, E_1^u \underline{\tilde{u}}, E_1^u \underline{\tilde{\tilde{u}}}\right) \\ \vdots \\ \zeta_{n_{sys}}\left(\tau, E_{n_{sys}}^u \underline{\tilde{u}}, E_{n_{sys}}^u \underline{\tilde{\tilde{u}}}\right) \end{pmatrix} \end{cases} \tag{25}$$

Ψ_τ was referred as Ψ in [6] and did not include the derivatives in its formulation, yet the smoothness enhancement done by the Hermite interpolation led to an underlying use of these derivatives.

When the result of the mth iteration is available, a fixed-point iteration on macro-step $\tau = [t^{[N]}, t^{[N+1]}[$ is simply done by:

$$\begin{pmatrix} {}^{[m+1]}\underline{\tilde{u}} \\ {}^{[m+1]}\underline{\tilde{\tilde{u}}} \end{pmatrix} := \Psi_\tau\left(\begin{pmatrix} {}^{[m]}\underline{\tilde{u}} \\ {}^{[m]}\underline{\tilde{\tilde{u}}} \end{pmatrix}\right) \tag{26}$$

3.3 First and last integrations of a step

The first iteration of a given macro-step $\tau \in \mathbb{T}$ is a particular case to be taken into account. Considering the breakdown presented in Sect. 2.5, this corresponds to case 1, case 2 subcase 2.2, case 3 first bullet point, and case 3 second bullet point when falling into subcase 2.2.

All these cases have something in common: they denote calls to ζ_k using a τ argument that has never been used in a previous call of ζ_k. In these cases, the latter function is defined by (21).

For this reason, the first call of γ_τ for a given macro-step τ will be completed before applying the JFM. Then, every time the JFM will call γ_τ, the $(\zeta_k)_{k\in[\![1,n_{sys}]\!]}$ functions called by γ_τ will behave the same way.

Once the JFM method ends, if it converged, a last call to γ_τ is made with the solution $\left(({}^{[m_{\max}(N)]}\underline{\tilde{u}}^{[N]})^T,\ ({}^{[m_{\max}(N)]}\underline{\tilde{\tilde{u}}}^{[N]})^T\right)^T$ for the systems to be in a good state for the next step (as explained in Sect. 2.5, the state of a system is hidden but affected at each call to a step function).

3.4 Step size control

The step size control is defined with the same *rule-of-thumbs* than the one used in [6]. The adaptation is not done on an error-based criterion such as in [13], but instead with a predefined rule based on the convergence of the iterative method (yes/no).

A minimal step size $\delta t_{\min} \in \mathbb{R}_*^+$, a maximal step size $\delta t_{\max} \in \mathbb{R}_*^+$ and an initial step size $\delta t_{\text{init}} \in [\delta t_{\min}, \delta t_{\max}]$ are defined for any simulation with IFOSMONDI-JFM method. At certain times (the communication times), the method will be allowed to reduce this step to help the convergence of the JFM.

The convergence criterion for the iterative method is defined by the rule (27).

Given $(\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}}) \in (\mathbb{R}_+^*)^2$,
convergence is reached when

$$\left|\gamma_\tau\begin{pmatrix} \underline{\tilde{u}} \\ \underline{\tilde{\tilde{u}}} \end{pmatrix}\right| < \left|\begin{pmatrix} \underline{\tilde{u}} \\ \underline{\tilde{\tilde{u}}} \end{pmatrix}\right| \varepsilon_{\text{rel}} + \left|\begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}\right| \varepsilon_{\text{abs}} \tag{27}$$

When the iterative method does not converge on the step $[t^{[N]}, t^{[N+1]}[$, either because a maximum number of iterations is reached or for any other reason (linear search does not converge, a Krylov internal method finds a singular matrix, ...), the step will be rejected and retried on the half (28) without subceeding $\delta t_{\min}$. Otherwise, once the method converged on $[t^{[N]}, t^{[N+1]}[$, the next integration step τ tries to increase the size of 30%, without exceeding $\delta t_{\max}$.

Once the iterative method exits on τ_{old}, the next step τ_{new} is defined by expression (28).

$$\tau_{\text{new}} = \begin{cases} \left[\sup(\tau_{\text{old}}), \min\left\{t^{\text{end}}, \sup(\tau_{\text{old}}) + \min\left\{\delta t_{\max}, 1.3\left(\sup(\tau_{\text{old}}) - \inf(\tau_{\text{old}})\right)\right\}\right\}\right[& \text{if convergence (27) was reached} \\ \left[\inf(\tau_{\text{old}}), \inf(\tau_{\text{old}}) + \max\left\{\delta t_{\min}, \frac{\sup(\tau_{\text{old}}) - \inf(\tau_{\text{old}})}{2}\right\}\right[& \text{otherwise (divergence)} \end{cases} \tag{28}$$

When $\varepsilon_{\text{abs}} = \varepsilon_{\text{rel}}$, these values will be denoted by ε.

When $\delta t_{\max} = \delta t_{\text{init}}$, these values will be denoted by δt_{ref}.

When the step size cannot be reduced as $\delta t_{\min}$ is reached, the co-simulation stops with an error. One can retry with a smaller $\delta t_{\min}$, or with $\delta t_{\min} = 0$.

4 Note on the implementation

Our implementation is based on an orchestrator-worker architecture, where $n_{sys} + 1$ processes are involved. One of them is dedicated to global manipulations: the *orchestrator*. It is not responsible of any system and only deals with global quantities (such as the time, the step τ, the $\underline{\tilde{u}}$ and $\underline{\tilde{y}}$ vectors and the corresponding time-derivatives, and so on). The n_{sys} remaining processes, the *workers*, are responsible for one system each. They only deal with local quantities related to the system they are responsible for.

4.1 Parallel evaluation of γ_τ using MPI

An evaluation of γ_τ consists in evaluations of the n_{sys} functions $(\zeta_k)_{k\in[\![1,n_{sys}]\!]}$, plus some manipulations of vectors and matrices (24). An evaluation of a single ζ_k for a given $k \in [\![1, n_{sys}]\!]$ consists in polynomial computations and an integration (21) (23) through a call of the corresponding $\hat{\hat{S}}_k$ function (13).

A single call to γ_τ can be evaluated parallelly by n_{sys} processes, each of them carrying out the integration of one of the systems. To achieve this, the MPI standard (standing for *Message Passing Interface* has been used, as the latter provides a routine to handle multi-process communications of data.

As the kth system only needs $E^u_k\underline{\tilde{u}}$ and $E^u_k\underline{\tilde{\dot{u}}}$ (see (3)) among $\underline{\tilde{u}}$ and $\underline{\tilde{\dot{u}}}$, the data can be send in an optimized manner from the orchestrator process to n_{sys} workers by using the `MPI_Scatterv` routine.

Analogously, each worker process will have to communicate their contribution both to the outputs and their derivatives (assembling the block vector at the right of the expression (24)). This can be done by using the `MPI_Gatherv` routine.

Finally, the communication of global quantities such as τ, m, the notifications of statuses and so on, can be done easily thanks to the `MPI_Broadcast` routine.

In all cases, the communications are organized in a "bus" architecture (all workers communicate with the orchestrator, but not to one another). Synchronization points before and after each evaluation of all ζ_k functions for all k in $[\![1, n_{sys}]\!]$ (in a single call of γ_τ) would generate, in the worst case (when every system has connection with every other system), $n_{sys}(n_{sys} - 1)$ communications for every input/output dispatching or gathering in a point-to-point architecture, whereas only $2\, n_{sys}$ communications are needed for a bus architecture, with the same total amount of exchanged data. Thus, our code uses the bus architecture.

4.2 Using PETSc for the JFM

PETSc [2, 3] is a library used for parallel numerical computations. For this paper, the several matrix-free versions of the Newton method and variants implemented in PETSc were very attractive. Indeed, the flexibility of this library at runtime enables the use of command-line arguments to control the resolution: `−snes_mf` orders the use of a matrix-free non-linear solver, `−snes_typenewtonls`, `anderson` [1] and `ngmres` [10] are various usable solving methods that can be used as JFMs, `−snes_atol`, `−snes_rtol` and `−snes_max_it` control the convergence criterion, `−snes_converged_reason`, `−snes_monitor` and `−log_view` produce information and statistics about the run, ...

This subsection proposes a solution to use these PETSc implementations in a manner that is compliant with the parallel evaluation of the JFM's callback (24). This implementation has been used to generate the results of Sect. 5.

First of all, PETSc needs a view on the environment of the running code: the processes, and their relationships. In our case, the $n_{sys} + 1$ processes of the orchestrator-worker architecture are not dedicated to the JFM. Thus, PETSc runs on the orchestrator process only. In terms of code, this can be done by creating PETSc objects referring to `PETSC_COMM_SELF` communicator on the orchestrator process, and creating no PETSc object on the workers.

The callback γ_τ implements internally the communications with the workers, and is given to the PETSc `SNES` object. The `SNES` non-linear solver will call this callback blindly, and the workers will be triggered *behind the scene* for integrations, preceded by the communications of the $\left(({}^{[m_{\max}(N)]}\underline{\tilde{u}}^{[N]})^T,\ ({}^{[m_{\max}(N)]}\underline{\tilde{\dot{u}}}^{[N]})^T\right)^T$ values asked by the `SNES` and followed by the gathering of the outputs and related derivatives. The latters are finally returned to PETSc by the callback on the orchestrator side, after reordering and dispatching them as in (24).

4.3 JFM's callback implementation

In this section, a suggestion of implementation is proposed for the γ_τ function, both on the orchestrator side and on the workers side. Precisions about variables in the snippets are given below them.

By convention, the process of rank 0 is the orchestrator, and any process of rank $k \in [\![1, n_{sys}]\!]$ is responsible of system k.

Snippet 1. JFM's callback on the orchestrator side (γ_τ)

```
PetscErrorCode JFM_callback(SNES /* snes */, Vec u_and_du, Vec res, void *
     ctx_as_void)
{
   MyCtxType *ctx = (MyCtxType*)ctx_as_void;
   const int order = DO_A_STEP;
   PetscScalar const * pscalar_u_and_du;
   PetscScalar * pscalar_res;
   size_t k;

   // conversion PetscScalar -> C double
   VecGetArrayRead(u_and_du, &pscalar_u_and_du);
   for (k = 0; k < ctx->n_in_tot * 2; k++)
      ctx->double_u_and_du[k] = (double)(pswork_x[k]);
   VecRestoreArrayRead(u_and_du, &pscalar_u_and_du);

   // Notify workers that we want them to run,
   // and telling them what tau is
   MPI_Bcast(&order, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(&(ctx->t_N),   1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(&(ctx->t_Np1), 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   // Apply extractors and communicate at the same time:
   // values, then derivatives
   MPI_Scatterv(ctx->double_u_and_du, ctx->in_sizes, ctx->in_offsets,
       MPI_DOUBLE, NULL, 0, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Scatterv(ctx->double_u_and_du + ctx->n_in_tot, ctx->in_sizes, ctx->
       in_offsets, MPI_DOUBLE, NULL, 0, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   /* Workers proceed integration here */

   // Assemble vector R^{\bar{y}} * (\zeta_1^T, ... \zeta_2^T)^T directly
   // while communicating values and derivatives
   MPI_Gatherv_outputs(MPI_IN_PLACE, 0, MPI_DOUBLE, ctx->work1_n_out_tot, ctx
       ->out_sizes, ctx->out_offsets, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Gatherv_outputs(MPI_IN_PLACE, 0, MPI_DOUBLE, ctx->work2_n_out_tot, ctx
       ->out_sizes, ctx->out_offsets, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   // Dispatching (equivalent of [[Phi^T, 0], [0, Phi^T]])
   dispatch(ctx->work1_n_out, ctx->out_sizes, ctx->n_sys,
      ctx->double_res, ctx->in_sizes, ctx->connections);
   dispatch(ctx->work2_n_out, ctx->out_sizes, ctx->n_sys,
      ctx->double_res + ctx->n_in_tot, ctx->in_sizes, ctx->connections);

   // Difference between original entries and permuted outputs
   for (k = 0; k < ctx->n_in_tot * 2; k++)
      ctx->double_res[k] = ctx->double_u_and_du[k] - ctx->double_res;

   // conversion C double -> PetscScalar
   VecGetArray(res, &pscalar_res);
   for (k = 0; k < ctx->n_in_tot * 2; k++)
      pswork_f[k] = (PetscScalar)(ctx->double_res[k]);
   VecRestoreArray(res, &pscalar_res);

   return 0;
}
```

On the worker's side, the corresponding running code section is the one in Snippet 2.

Snippet 2. JFM's callback on the worker side (ζ_k and communications)

```
/* ... */

while (1)
{
   MPI_Bcast(&order, 1, MPI_INT, 0, me__->comm);
   if (order != DO_A_STEP)
      break;

   // get tau
   MPI_Bcast(&t_N,   1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(&t_Np1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   // receive relevant inputs and derivatives for this system
   MPI_Scatterv(NULL, ctx->in_sizes, ctx->in_offsets, MPI_DOUBLE,
      sys_inputs, sys_n_in, MPI_DOUBLE, 0, me__->comm);
   MPI_Scatterv(NULL, ctx->in_sizes, ctx->in_offsets, MPI_DOUBLE,
      sys_dinputs, sys_n_in, MPI_DOUBLE, 0, me__->comm);

   /* integration: */
   zeta_do_a_step(t_N, t_Np1, inputs, sys_dinputs, // [in]
      sys_outputs, sys_doutputs); // [out]

   // send the outputs and derivatives (results of zeta)
   MPI_Gatherv_outputs(sys_outputs, sys_n_out, MPI_DOUBLE,
      NULL, NULL, NULL, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Gatherv_outputs(sys_doutputs, sys_n_out, MPI_DOUBLE,
      NULL, NULL, NULL, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

/* ... */
```

The aim is not to show the code that has been used to generate the results of Sect. 5, but to figure out how to combine the PETSc and MPI standard (PETSc being based on MPI) to implement a parallel evaluation of γ_τ.

In the code snippet 1, the function `JFM_callback` is the one that is given to the PETSc `SNES` object with `SNESSetFunction`. The context pointer `ctx` can be anything that can be used to have access to extra data inside of this callback. The principle is: when `SNESSolve` is called, the callback function which has been given to the `SNES` object will be called an unknown number of times. For this example, we suggested a context structure `MyCtxType` at least containing:

- `t_N`, `t_Np1` the boundary times of τ, *id est* $t^{[N]}$ and $t^{[N+1]}$ (as `double` each),
- `n_in_tot` the total number of inputs $n_{in,tot}$ (as `size_t`),
- `double_u_and_du` an array dedicated to the storage of $(\underline{\tilde{u}}^T, \underline{\tilde{\dot{u}}}^T)^T$ (as `double` $*$),
- `in_sizes` the array containing the number of inputs for each process $(n_{in,k})_{k\in[\![0,n_{sys}]\!]}$ including process 0 (with the convention $n_{in,0} = 0$) (as `int` $*$),
- `in_offsets` the memory displacements $\left(\sum_{l=1}^{k} n_{in,l}\right)_{k\in[\![0,n_{sys}]\!]}$ for inputs scattering for each process (as `int` $*$),
- `work1_n_out_tot` and `work2_n_out_tot` two arrays of size $n_{out,tot}$ for temporary storage (as `double` $*$),
- `out_sizes` and `out_offsets` two arrays analogous to `in_sizes` and `in_offsets` respectively, considering the outputs,
- `n_sys` tot number of systems n_{sys} (as `size_t`),
- `double_res` an array of size 2 $n_{in,tot}$ dedicated to the storage of the result of γ_τ (as `double` $*$), and
- `connections` any structure to represent the connections between the systems $\boldsymbol{\Phi}^T$ (a full matrix might be a bad idea as $\boldsymbol{\Phi}$ is expected to be very sparse).

The function `dispatch` is expected to process the dispatching (15) of the values given in its first argument into the array pointed by its fourth argument.

Please note that the orchestrator process has to explicitly send an order different from `DO_A_STEP` (with `MPI_Bcast`) to notify the workers that the callback will not be called anymore. Nonetheless, this order might not be sent right after the call to `SNESSolve` on the orchestrator side. Indeed, if the procedure converged, a last call has to be made explicitly in the orchestrator (see Sect. 3.3).

Another explicit call to `JFM_callback` should also be explicitly made on the orchestrator side before the call of `SNESSolve` (as also explained in Sect. 3.3).

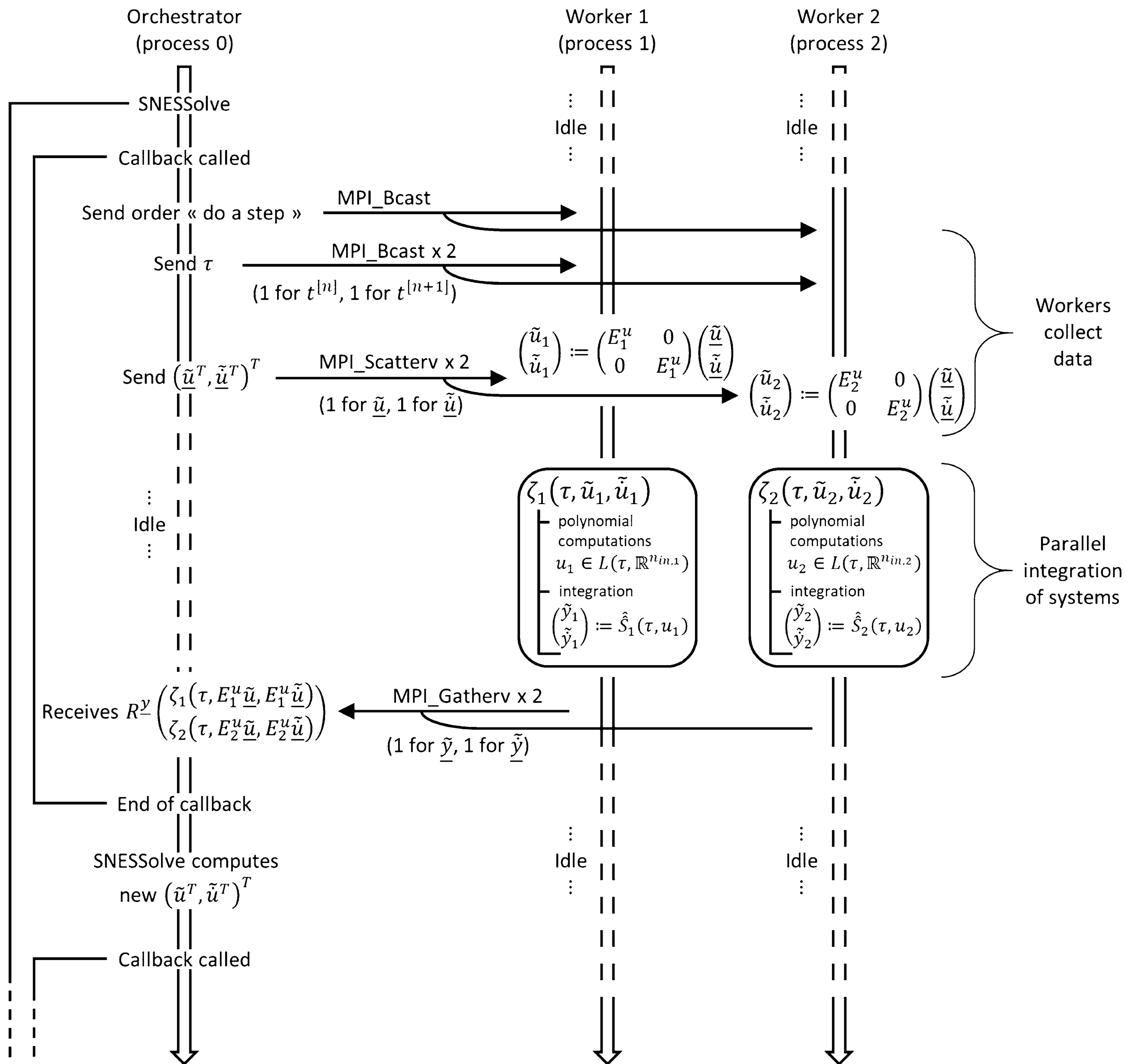

Fig. 7 Workflow of the callback function called by SNESSolve: example with $n_{sys} = 2$ (external first call to the callback is supposed to be already made before `SNESSolve` is called)

Figure 7 presents a schematic view of these two snippets running parallelly.

5 Results on test cases

Two test cases will be treated here. The first one is a simple case that enables to understand the kind of configurations that really benefit from the IFOSMONDI-JFM method (*id est*when the function of the fixed-point formulation is not contractant), and the second one is an industrial-scale model with 148 interface variables that allows the comparison of classical IFOSMONDI (based on the fixed-point method), IFOSMONDI-JFM and the natural explicit ZOH co-simulation method in terms of time/accuracy trade-off.

5.1 Mechanical model with multiple feed-through

Difficulties may appear in a co-simulation problem when the coupling is not straightforward. Some of the most difficult cases to solve are the algebraic coupling (addressed in [8]) arising from causal conflicts, and the multiple feed-through,

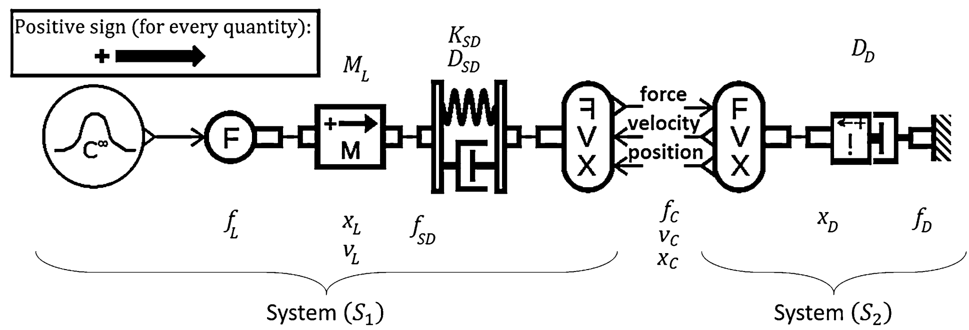

Fig. 8 Mass spring damper with damping reaction modelled with Simcenter Amesim - Parameters are above, variables are below

*id est*the case where outputs of a system linearly depend on its inputs, and the connected system(s) have the same behavior. In some case, this may lead to a non-contractant Ψ_τ function.

This section presents a test case we designed, belonging to this second category. The fixed-point convergence can be accurately analyzed so that its limitations are highlighted.

Please note that this test case is intentionally simple in order to easily enlight the enhancements brought by the IFOSMONDI-JFM method compared to the fixed-point IFOSMONDI method. Although very simple, this example enables to understand the convergence properties of the proposed JFM, as the latter is not objected by the non-contractance of Ψ_τ (contrary to a fixed-point underlying method like in classical IFOSMONDI).

5.1.1 Test case presentation

The test case has been modeled, parameterized and simulated with Simcenter Amesim software, a 0D modeling and simulation software developed by Siemens Industry Software. The co-simulations have been run with our code (implementing fixed-point IFOSMONDI and IFOSMONDI-JFM algorithms), coupled with the systems modeled in Simcenter Amesim for underlying $\hat{\hat{S}}_k$ evaluations (see Fig. 2) in ζ_k evaluation (polynomial input computations stage in ζ_k happens in our code).

Figure 8 represents a 1-mass test case with a classical mechanical coupling on force, velocity and position. These coupling quantities are respectively denoted by f_c, v_c and x_c. The component on the right represents a damper with a massless plate, computing a velocity (and integrating it to compute a displacement) by reaction to a force input.

We propose the parameters values in Table 1.

All variables will be denoted by either f, v or x (corresponding to forces, velocities and positions, respectively) with an index specifying its role in the model (see Fig. 8).

The predefined force f_L is a C^∞ function starting from 5 N and definitely reaching 0 N at $t = 2$ s. The expression of f_L is (29) and the visualization of it is presented on Fig. 9.

Table 1 Parameters and initial values of the test case model

Notation	Description	Value
M_L	Mass of the body in (S_1)	1 kg
K_{SD}	Spring rate of the spring in (S_1)	1 N/m
D_{SD}	Viscosity coefficient in the damper in (S_1)	1 N/(m/s)
D_D	Viscosity coefficient in the damper in (S_2)	$\in [0.01, 4]$
$x_L(0)$	Initial position of the body in (S_1)	0 m
$v_L(0)$	Initial velocity of the body in (S_1)	0 m/s
$x_D(0)$	Initial position of the plate in (S_2)	0 m
t^{init}	Initial time	0 s
t^{end}	Final time	10 s

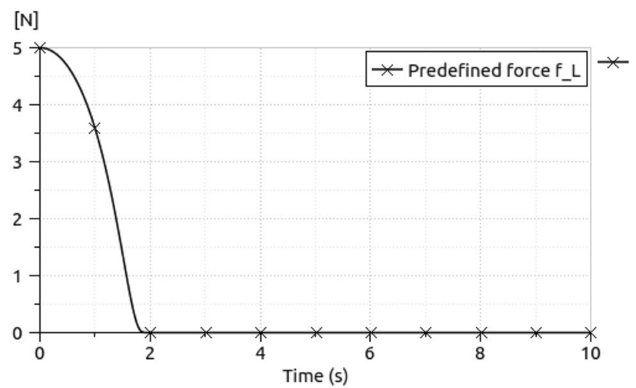

Fig. 9 Predefined force f_L

$$f_L : \begin{cases} [0,10] & \rightarrow [0,5] \\ t & \mapsto \begin{cases} \dfrac{5}{e^{-1}} e^{\left(\left(\frac{t}{2}\right)^2 - 1\right)^{-1}} & \text{if } t < 2 \\ 0 & \text{if } t \geqslant 2 \end{cases} \end{cases} \tag{29}$$

The expected behavior of the model is presented in Table 2 referring to conventionnal directions of Fig. 10.

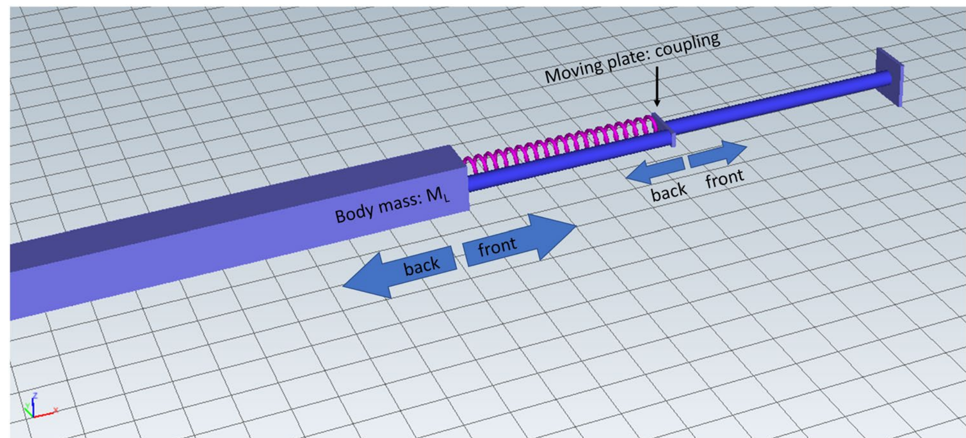

Fig. 10 Test model visualized with Simcenter Amesim

Table 2 Main stages of a simulation of the test case model

Stage	Body displacement	Plate displacement	Description
1	Front	Front	Positive f_L pushes everything
2	Back	Front	The spring pushes the body backward as it is close to the plate
3	Back	Back	The spring pulls the plate backwards as the body is moving backward with inertia
4	Front	Back	The spring pulls the body forward as the inertia made it go too far in the backward direction
5	Front	Front	The body is still moving frontward with inertia, so the compressed spring pushes the plate forward

The behavior presented in Table 2 might slightly change while parameter D_D changes (all other parameters being fixed, see Table 1).

5.1.2 Equations and eigenvalues of the fixed-point callback Ψ_τ

The displacement of the mass M_L is due to the difference of the forces applied on its left side (f_L, generated from a force source, cf. Fig. 9) and on its right side (f_{SD}, resulting from spring compression/dilatation and damper effect). This movement can be computed using the acceleration of the mass. Indeed, second Newton's law gives:

$$\begin{aligned} \dot{v}_L &= (f_L + f_{SD}) M_L^{-1} \\ \dot{x}_L &= v_L \end{aligned} \tag{30}$$

and the spring and damper forces can be expressed the following way:

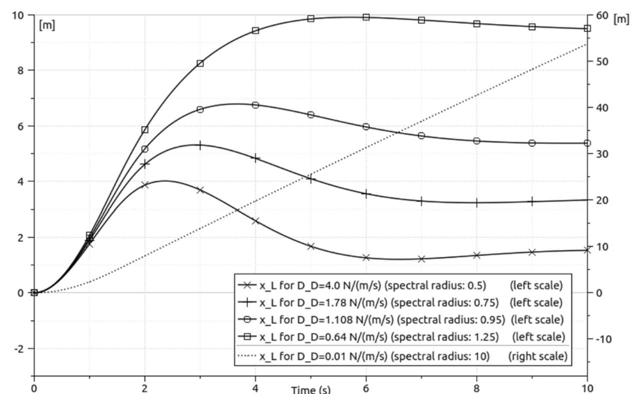

Fig. 11 Displacement of the mass (x_L) for different damping ratios of the right damper (D_D) simulated on a monolithic model (without co-simulation). Associated spectral radii of J_Ψ are recalled for further coupled formulations

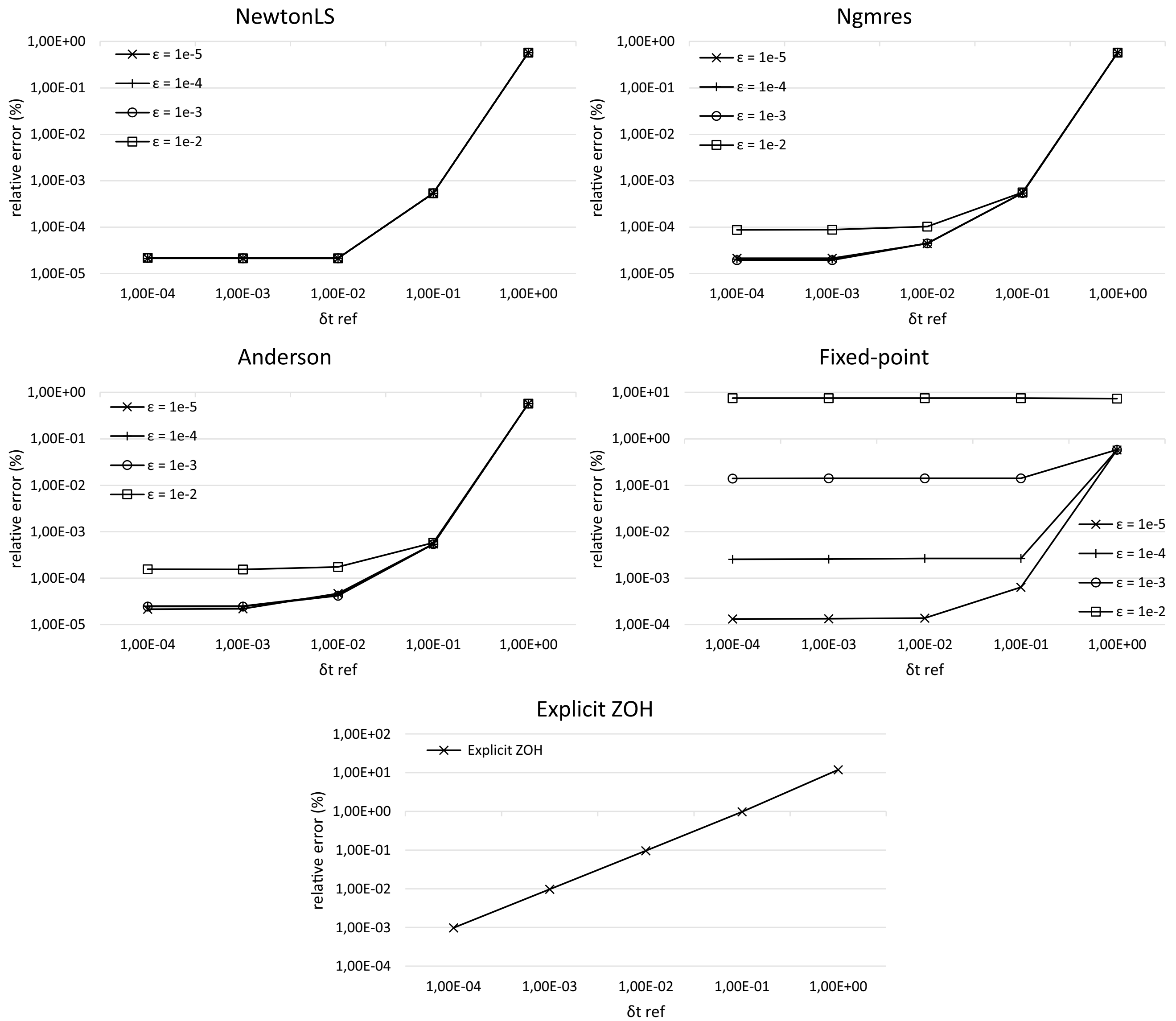

Fig. 12 Error accross δt_{ref} with different methods on a contractant case ($D_D = 4.0,\ \rho(J_{Psi}) = 0.5$)—NewtonLS, Ngmres and Anderson are matrix-free iterative methods used with the IFOSMONDI-JFM algorithm, Fixed-point is the fixed-point IFOSMONDI algorithm, and Explicit ZOH is the non-iterative zero-order hold fixed-step co-simulation

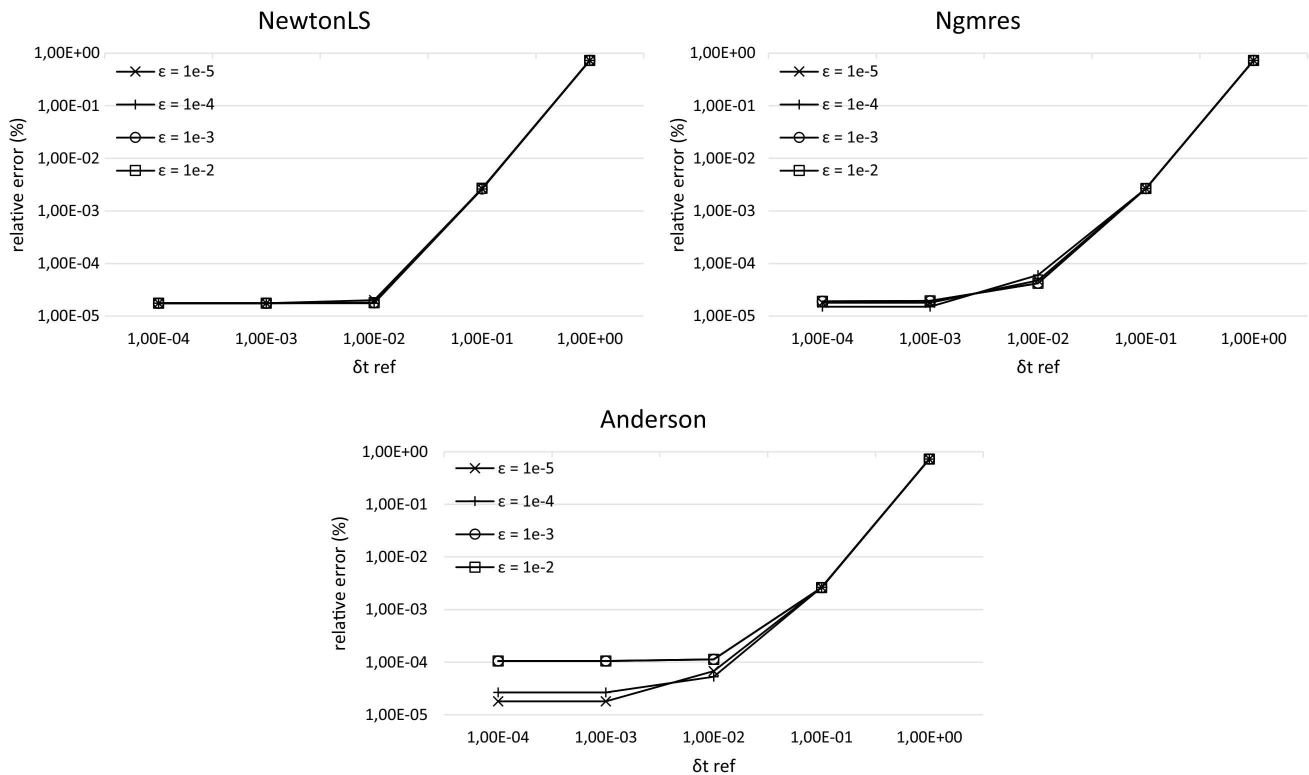

Fig. 13 Error accross δt_{ref} with different methods on a non-contractant case ($D_D = 0.64$, $\rho(J_{Psi}) = 1.25$)—NewtonLS, Ngmres and Anderson are matrix-free iterative methods used with the IFOSMONDI-JFM algorithm

$$\begin{aligned} f_{SD} &= K_{SD}(x_C - x_L) + D_{SD}(v_C - v_L) \\ f_C &= -f_{SD} \\ f_D &= -D_D(0 - v_C) \\ f_C &= f_D \\ v_C &= {}^{f_C}/_{D_D} \end{aligned} \tag{31}$$

leading to the expression (32) of the coupled systems.

At a given time t, we can state the jacobian of Ψ_τ introduced in (25) using the expressions of the coupling quantities (32).

$$(S_1):\left\{\begin{aligned} \begin{pmatrix} \dot{v}_L \\ \dot{x}_L \end{pmatrix} &= \begin{pmatrix} \frac{-D_{SD}}{M_L} & \frac{-K_{SD}}{M_L} \\ 1 & 0 \end{pmatrix}\begin{pmatrix} v_L \\ x_L \end{pmatrix} + \begin{pmatrix} \frac{D_{SD}}{M_L} & \frac{K_{SD}}{M_L} \\ 0 & 0 \end{pmatrix}\begin{pmatrix} v_C \\ x_C \end{pmatrix} + \begin{pmatrix} \frac{f_L}{M_L} \\ 0 \end{pmatrix} \\ f_C &= \begin{pmatrix} D_{SD} & K_{SD} \end{pmatrix}\begin{pmatrix} v_L \\ x_L \end{pmatrix} + \begin{pmatrix} -D_{SD} & -K_{SD} \end{pmatrix}\begin{pmatrix} v_C \\ x_C \end{pmatrix} \end{aligned}\right. .$$

$$(S_2):\left\{\begin{aligned} \dot{x}_D &= 0\, x_D + \frac{1}{D_D} f_C \\ \begin{pmatrix} v_C \\ x_C \end{pmatrix} &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} x_D + \begin{pmatrix} \frac{1}{D_D} \\ 0 \end{pmatrix} f_C \end{aligned}\right. \tag{32}$$

Indeed, the output variables got at a call are at the same time than the one at which the imposed inputs are reached (end of the macro-step) thanks to the definitions of ζ_k.

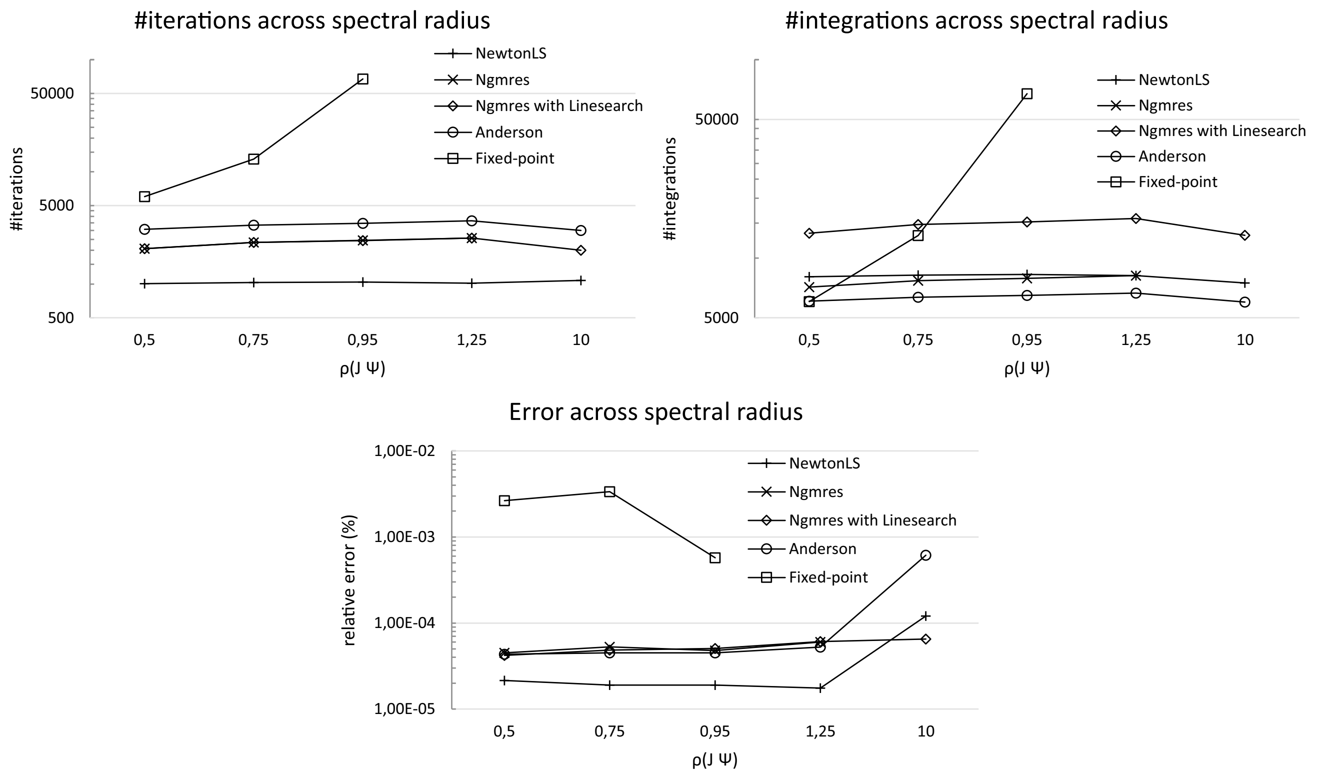

Fig. 14 Total number of iterations, integrations, and error across spectral radius of J_{Ψ} for different methods (Fixed-point corresponds to classical IFOSMONDI algorithms, and all other methods are used with the IFOSMONDI-JFM version). All co-simulation ran with $\varepsilon = 10^{-4}$ and $\delta t_{\text{ref}} = 10^{-2}$

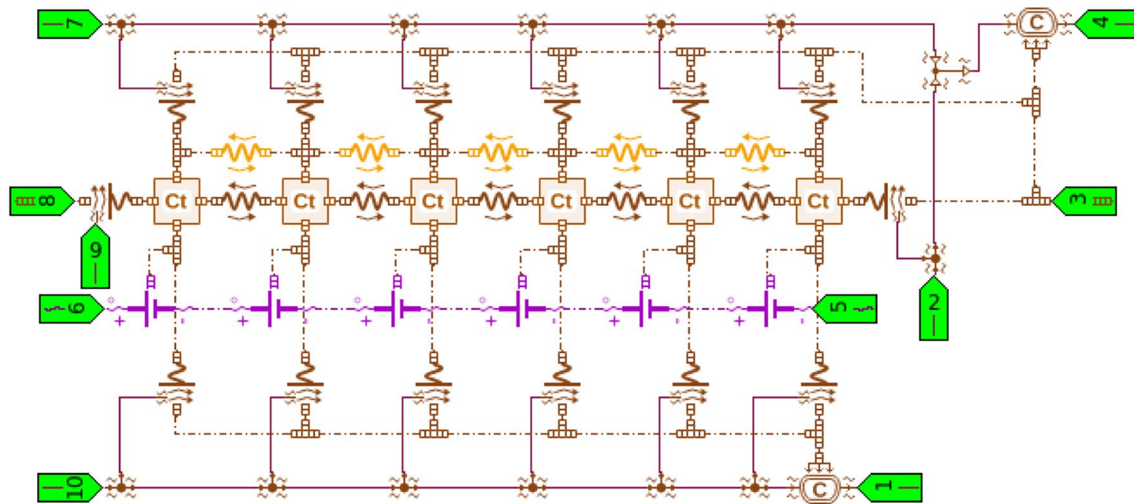

Fig. 15 Subsketch inside of a single module of the battery pack: the 6 cells can be seen

$$J_{\Psi_\tau}\left(\begin{pmatrix} f_C \\ v_C \\ x_c \\ \dot{f_C} \\ \dot{v_C} \\ \dot{x_c} \end{pmatrix}\right) = \left(\begin{array}{ccc|ccc} \boxed{0} & -D_{SD} & -K_{SD} & \boxed{0} & 0 & 0 \\ {}^1/_{D_D} & \boxed{0} & \boxed{0} & 0 & \boxed{0} & \boxed{0} \\ 0 & \boxed{0} & \boxed{0} & 0 & \boxed{0} & \boxed{0} \\ & & & \boxed{0} & -D_{SD} & -K_{SD} \\ & \text{Block} & & {}^1/_{D_D} & \boxed{0} & \boxed{0} \\ & & & 0 & \boxed{0} & \boxed{0} \end{array}\right) \tag{33}$$

The framed zeros are “by-design” zeros: indeed, systems never produce outputs depending on inputs given to other systems. The block called “Block” in (33) depends on the methods used to retrieve the time-derivatives of the coupling quantities (see (13) and its finite differences version). Nevertheless, this block does not change the eigenvalues of J_{Ψ_τ}

as it is a block-triangular matrix. Indeed, the characteristic polynomial of $I_6 - \lambda J_{\Psi_\tau}$ is the product of the determinant of the two 3×3 blocks on the diagonal of $I_6 - \lambda J_{\Psi_\tau}$. The eigenvalues of J_Ψ are:

$$0, \ +1i\sqrt{\frac{D_{SD}}{D_D}}, \ -1i\sqrt{\frac{D_{SD}}{D_D}} \qquad \text{(each with a multiplicity of 2)} \tag{34}$$

Hence, the following relation between the parameters and the spectral radius can be shown (given $D_D > 0$ and $D_{SD} = 1 > 0$):

$$\varrho(J_{\Psi_\tau}) \begin{cases} < 1 & \text{if } D_{SD} < D_D \\ \geqslant 1 & \text{if } D_{SD} \geqslant D_D \end{cases} \tag{35}$$

We can thus expect that the fixed-point IFOSMONDI co-simulation algorithm based on a fixed-point method [6] cannot converge on this model when the damping ratio of the component on the right of the model (see Fig. 8) is smaller than the damping ratio of the spring-damper component.

We will process several simulations with different values of D_D leading to different values of $\varrho(J_{\Psi_\tau})$. These values and the expected movement of the body of the system is plotted in Fig. 11.

5.1.3 Results

As the PETSc library enables to easily change the parameters of the JFM (as explained in Sect. 4.2), three methods have been used in the simulations:

- NewtonLS: a Newton based non-linear solver that uses a line search,
- Ngmres: the non-linear generalized minimum residual method [10], and
- Anderson: the Anderson mixing method [1]

First of all, simulations have been processed with all these JFMs (with parameters exhaustively defined in appendix A) within IFOSMONDI-JFM, the fixed-point IFOSMONDI algorithm (denoted hereafter as “Fixed-point”), and the original explicit zero-order hold co-simulation method (sometimes referred to as non-iterative Jacobi). The error is defined as the mean of the normalized L^2 errors on each state variable of both systems on the whole $[t^{\text{init}}, t^{\text{end}}]$ domain. The reference is the monolithic simulation (of the non-coupled model) done with Simcenter Amesim. Such errors are presented for a contractant case ($D_D = 4$ N, so $\varrho(J_{\Psi_\tau}) = 0.5$) in Fig. 12. For a non-contractant case ($D_D = 0.64$ N, so $\varrho(J_{\Psi_\tau}) = 1.25$), analog plots are presented in Fig. 13.

As expected, the simulations failed (diverged) with fixed-point method for the non-contractant case. Moreover, the values given to the system were too far from physically-possible values with the explicit ZOH co-simulation algorithm, so the internal solvers of systems (S_1) and (S_2) failed to integrate. These are the reason why these two methods lead to no curve on Fig. 13.

Nonetheless, the three versions of IFOSMONDI-JFM algorithm keep producing reliable results with an acceptable relative error (less than 1%) when $\delta t_{\text{ref}} \geqslant 0.1$ s.

On Figs. 12 and 13, IFOSMONDI-JFM method seems to solve the problem with a good accuracy regardless of the value of the damping ratio D_D. To confirm that, several other values have been tried: the ones for which the solution has been computed and plotted in Fig. 11. The error is presented, but also the number of iterations and the number of integrations (calls to ζ_k, i.e. calls to γ_τ for IFOSMONDI-JFM or to Ψ_τ for fixed-point IFOSMONDI). Although for the fixed-point IFOSMONDI the number of iteration is the same than the number of integration, for the IFOSMONDI-JFM algorithm the number of iterations is the one of the underlying non-linear solver (NewtonLS, Ngmres or Anderson), and there might be a lot more integrations than iterations of the non-linear method. These results are presented in Fig. 14.

As expected, the threshold of $\varrho(J_{\Psi_\tau}) = 1$ (*id est* $D_D = D_{SD} = 1$) is critical for the fixed-point method. The IFOSMONDI-JFM method not only can overpass this threshold, but no significant extra difficulty appears to solve the problem in the non-contractant cases, except for the Ngmres non-linear solver (which failed to converge with $D_D = 0.01$, so with $\varrho(J_{\Psi_\tau}) = 10$). However, regarding the Ngmres method, the variant that uses line search converges in all cases. Even though the latter requires more integrations than other JFMs, it is more robust to high values of $\varrho(J_{\Psi_\tau})$. The parameters of this line search are detailed on Table 9 in appendix A.

The NewtonLS and Anderson methods show a slightly bigger error on this “extreme” case of $\varrho(J_{\Psi_\tau}) = 10$, yet it stays under 0.001% which is completely acceptable.

Among those two JFMs (NewtonLS and Anderson), the trend that can be observed on Fig. 14 shows that NewtonLS is always more accurate than Anderson, yet it always requires a bigger amount of integrations. We can stand that IFOSMONDI-JFM is more *accuracy-oriented* on this model when it is based on the NewtonLS JFM, and more *speed-oriented* on this model when it is based on the Anderson JFM (for the same δt_{ref} and ε). For high values of $\varrho(J_{\Psi_\tau})$, *accuracy-oriented* simulations are achieved thanks to the Ngmres JFM with line search more than the NewtonLS one.

Finally, smaller errors are obtained with IFOSMONDI-JFM and with less iterations than fixed-point IFOSMONDI. Yet, the time consumption is directly linked with the number of integrations, not with the number of iterations of the underlying non-linear solver. The total number of integrations does not increase across the problem difficulty

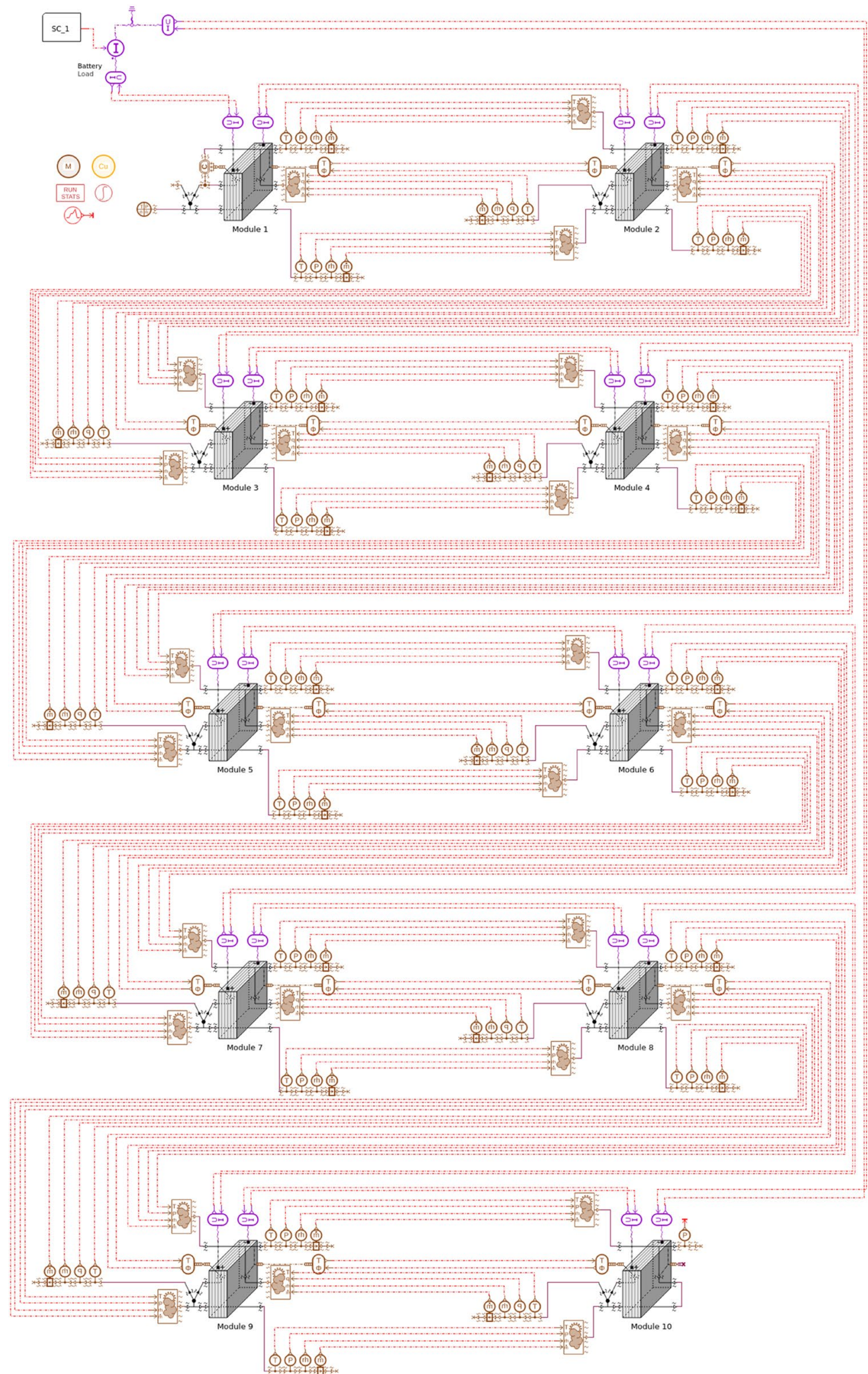

Fig. 16 Battery pack cooling system modelled with Simcenter Amesim (each module contains 6 cells as shown on Fig. 15)—Monolithical model

(increasing with $\varrho(J_{\Psi_\tau})$), and the non-linear methods within IFOSMONDI-JFM do not even require more integrations that the fixed-point one for most of the values of D_D for which the fixed-point IFOSMONDI algorithm does not fail.

5.2 Industrial-scale thermal-electric model

Regarding industrial-scale test cases, it is not always possible to determine in advance if the fixed-point formulation

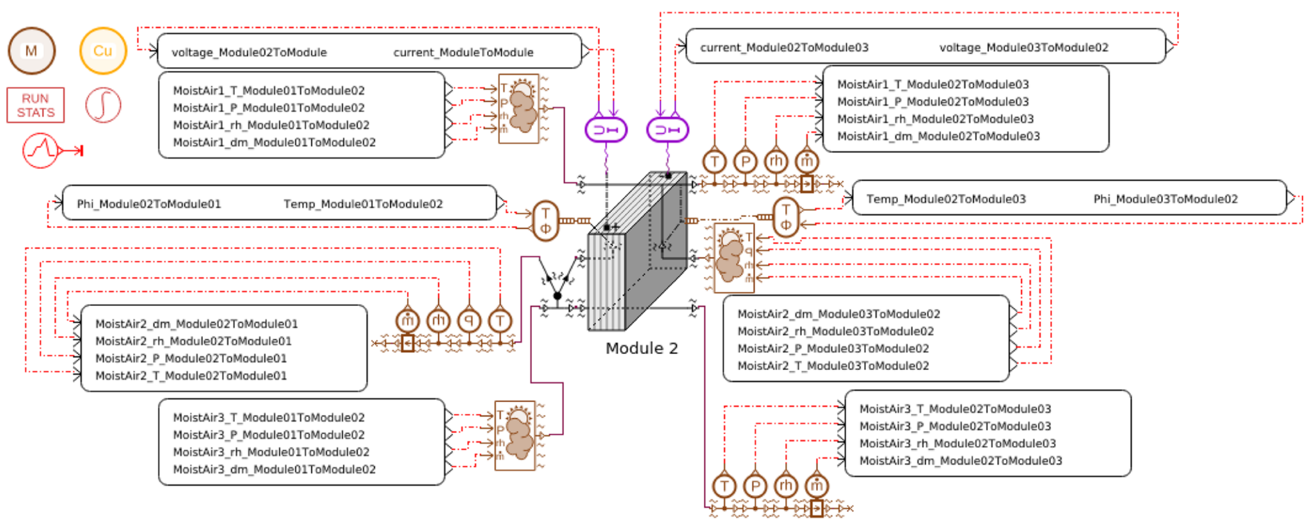

Fig. 17 Black-box system of module 2 only: sketch representation of a single system for co-simulation in Simcenter Amesim

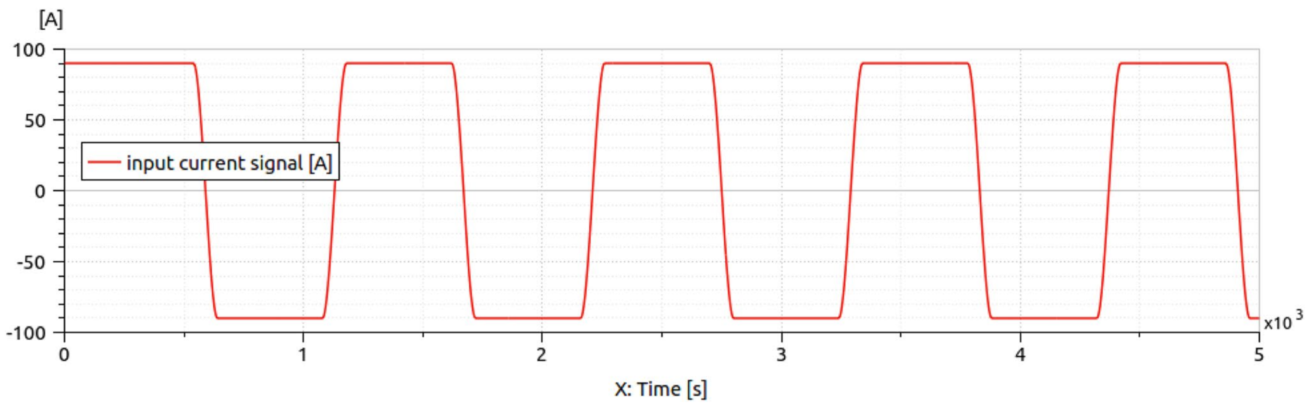

Fig. 18 Battery load/unload signal

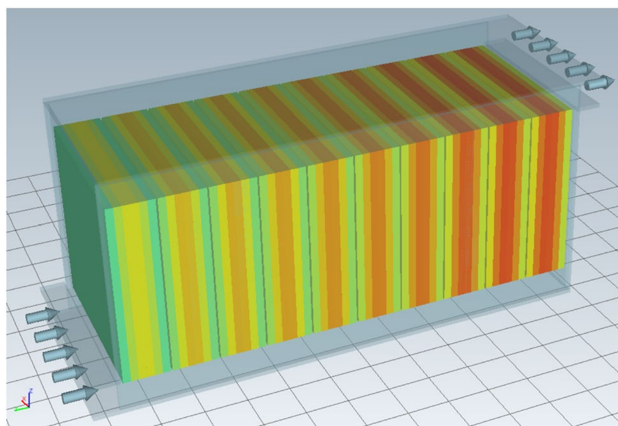

Fig. 19 Battery temperature distribution at $t = t^{\text{end}} = 5000$ —Arrows represent the air flow—Module 1 is on the left and module 10 is on the right

Table 3 Results on the Battery pack cooling system with IFOSMONDI-JFM

	IFOSMONDI-JFM (Anderson)			
	$\varepsilon = 10^{-8}$	$\varepsilon = 10^{-6}$	$\varepsilon = 10^{-4}$	$\varepsilon = 10^{-2}$
Error (in %)	0.001	0.0016	0.0017	0.0035
Elapse time	52′24″	12′46″	11′28″	5′27″
#iterations	347 189	68 015	56 464	23 229
#integrations	366 702	83 107	71 536	38 301
Average step size (s)	0.831	0.994	0.995	0.995
#rejected steps	734	4	0	0

is contractant or not. Indeed, the analytical analysis (as done for the first test-case) is not always possible due to the model dimensions and its potential non-linear behavior.

Table 4 Results on the Battery pack cooling system with fixed-point IFOSMONDI

	Classical IFOSMONDI (fixed-point)			
	$\varepsilon = 10^{-8}$	$\varepsilon = 10^{-6}$	$\varepsilon = 10^{-4}$	$\varepsilon = 10^{-2}$
Error (in %)	0.0017	0.0017	0.0046	0.07
Elapse time	17′10″	10′31″	5′00″	2′37″
#integrations	103 951	65 350	30 176	17 433
Average step size (s)	0.986	0.995	0.995	0.995
#rejected steps	47	0	0	0

Table 5 Results on the Battery pack cooling system with Explicit ZOH

	Explicit ZOH				
	$\delta t = 10^{-3}$	$\delta t = 10^{-2}$	$\delta t = 10^{-1}$	$\delta t = 1$	$\delta t = 10$
Error (in %)	0.0016	0.0154	0.154	1.806	15.047
Elapse time	10h23′03″	1h01′35″	9′05″	1′09″	12″
#integrations	5 000 000	500 000	50 000	5 000	500
Average step size (s)	10^{-3}	10^{-2}	10^{-1}	1	10

For this reason, this subsection introduces a large model with 324 state variables across eleven systems, and 148 interface variables in total (meaning 148 inputs connected to 148 outputs in a one-to-one way, making $\boldsymbol{\Phi}^T$ a square matrix). As the variable of the JFM is the vector of all inputs and their derivatives, the JFM solves a problem of size 296.

The problem is compliant with the fixed-point IFOSMONDI and the explicit ZOH methods, so that comparisons in terms of time/accuracy trade-off can be conducted. This analysis is namely possible thanks to the scale of the system, making it run non-instantaneously.

5.2.1 Model presentation

The model in an industrial-scale thermal-electric system representing a battery pack (represented on Fig. 16) with an air cooling system. The battery pack is made of 10 modules of 6 cells each (see Fig. 15), all modules being connected by several moist airports (to represent airflow as different points in space as the air is circulating), thermal connections (representing thermal conduction) and electrical connections.

In practice, the need for a co-simulation for this kind of model arises when an external tool (simulation and modelling platform) provides a black-box system for each module. Indeed, in this case, doing a co-simulation is the only way to test the battery pack made up of these modules (in a flexible configuration regarding the number of modules) regarding a given battery load/unload scenario.

In this paper, the monolithic system of Fig. 16 will only act as a reference, and we will consider 11 black-box systems respectively corresponding to the 10 modules and the external load/unload scenario. The sketch of one of the black-box module systems is given as example in Fig. 17, and the load/unload scenario (in the 11th system) is presented in Fig. 18.

The battery pack is a 230 V, 10.4 kWh hybrid vehicle battery. The cells in each module are 3.84 V, 45 Ah Li-Ion cells. The charge and discharge (smooth) steps that can be seen on Fig. 18 simulate critical cses where the highest thermal load occur (as the battery is submitted to high currents). The pack in a 20°C air environment. Air the airflow (cooling system) comes from the bottom of module 1 and exits the pack at the top of module 10, the temperature is distributed along with the modules and cells like as shown in Fig. 19 at the

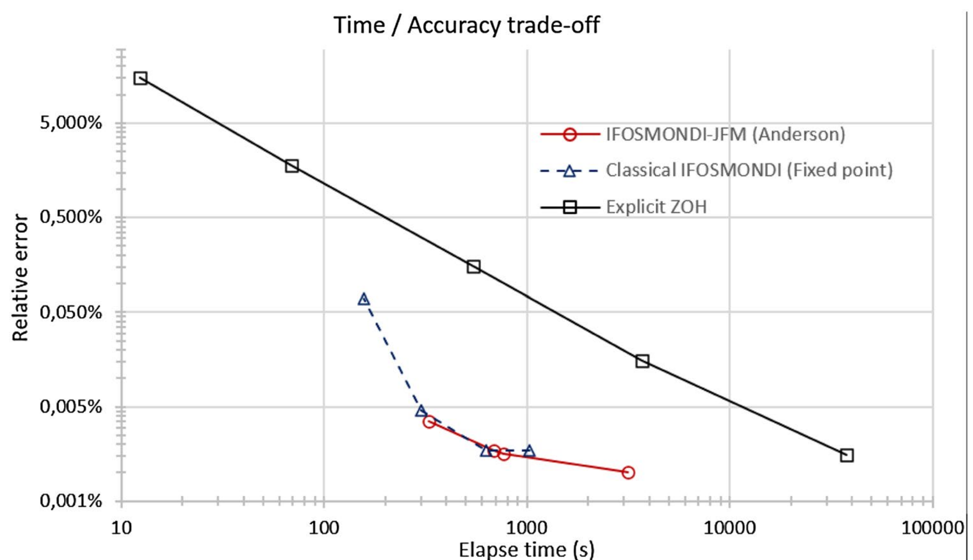

Fig. 20 Graphical visualization of results in Tables 3, 4 and 5

end of the 5000 s scenario. This result is obtained with the monolithical reference model.

5.2.2 Results

Results have been generated on a HPC cluster so that the processes can run in parallel. Indeed, due to the 10 modules of the model and the system containing the scenario, the battery load, and the reference potential, 11 workers are instantiated for a co-simulation. In addition to the orchestrator process (see architecture on Fig. 7), a total of 12 processes run parallelly.

Due to the small number of integrations required by the Anderson method (as it recombines the previously evaluated

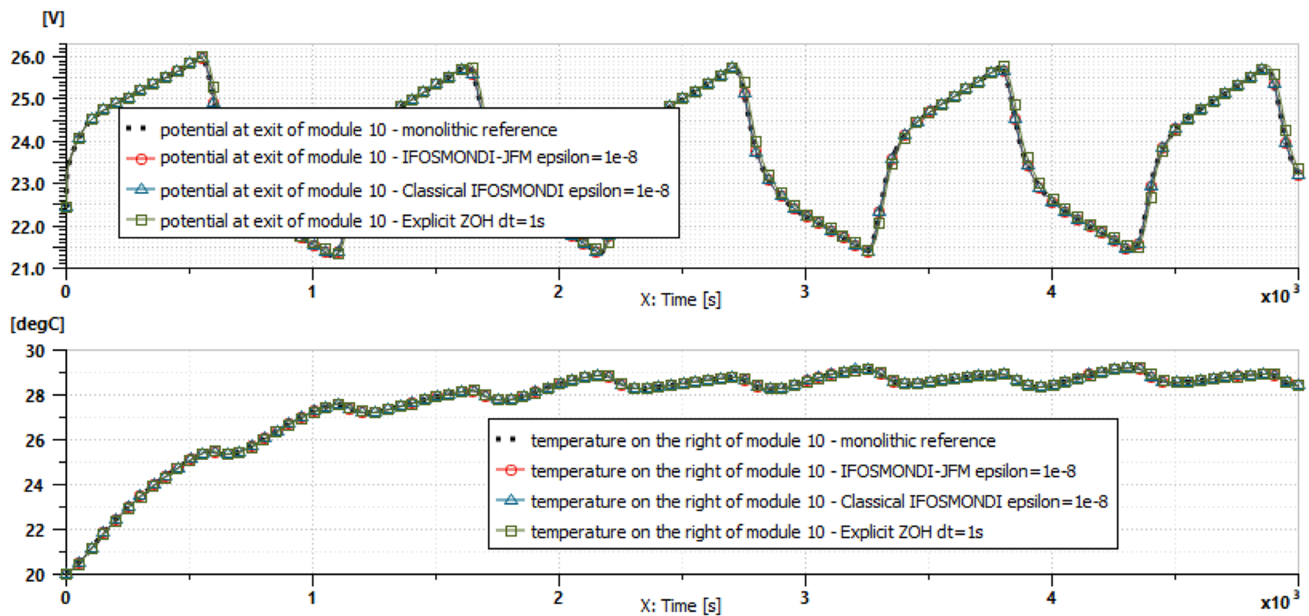

Fig. 21 Two variables of interest in the Battery Pack Cooling model, results for different (co-)simulation methods

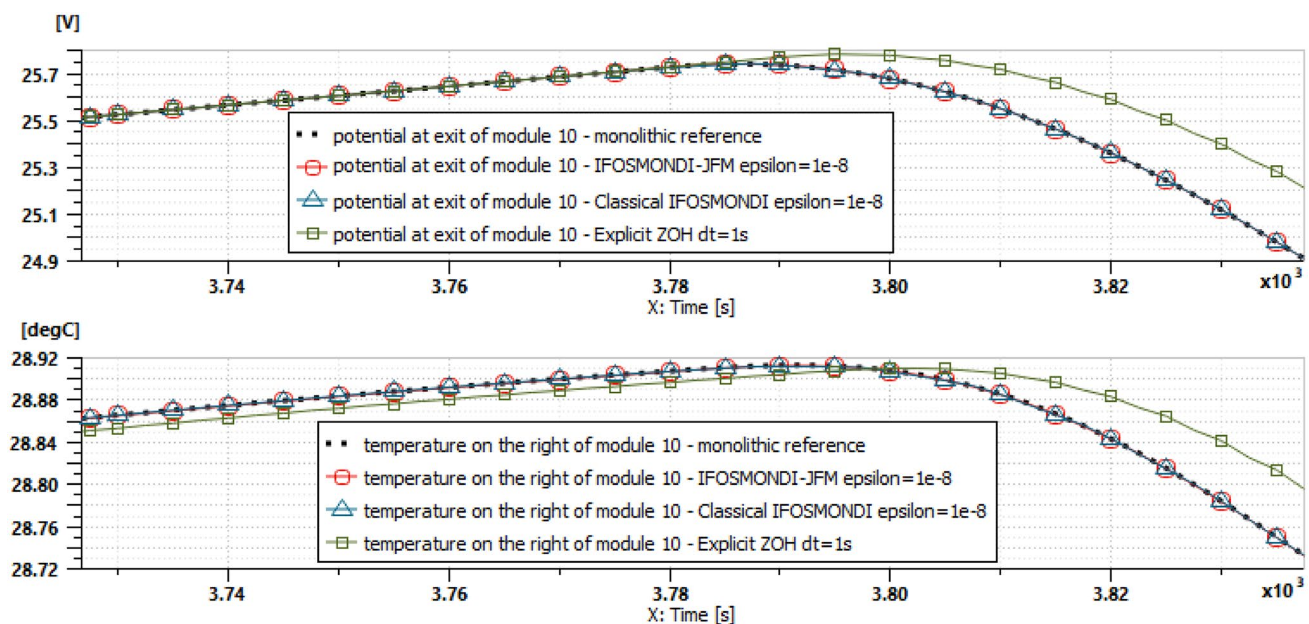

Fig. 22 Focus on $t \in [3730, 3830]$ of the curves in Fig. 21

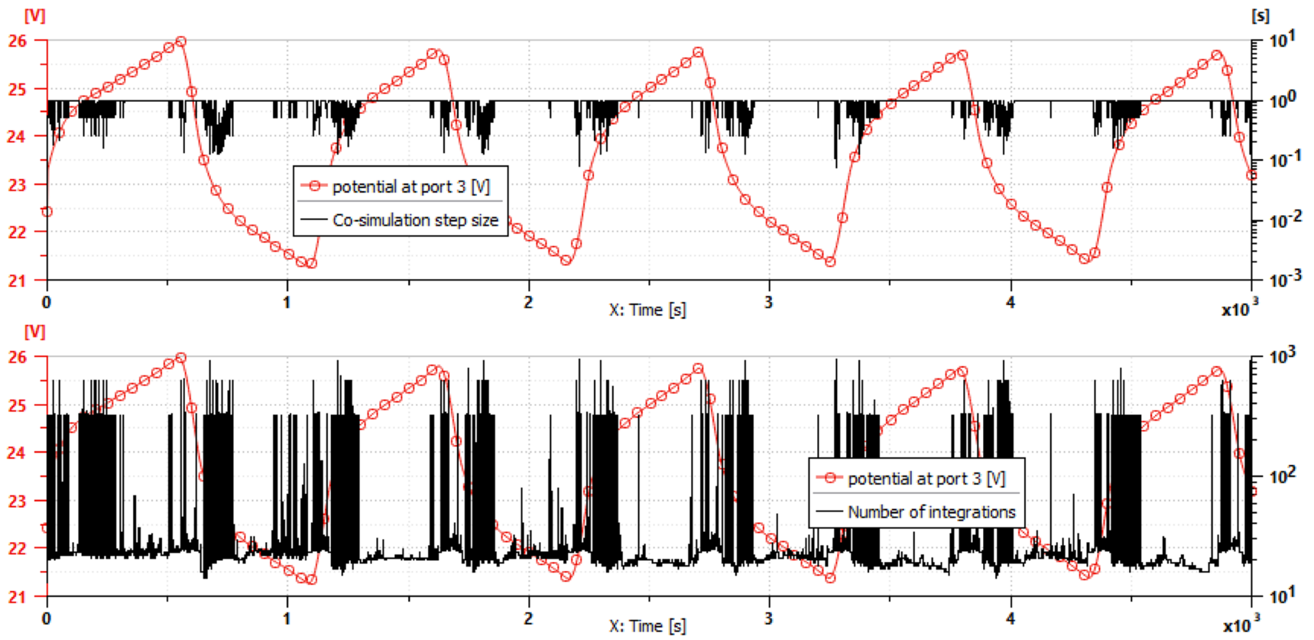

Fig. 23 Visualization of the connection between the co-simulation step size (upper straight curve, right y-scale), the number of integrations (lower straight curve, right y-scale) and a representative variable of interest of the system (superimposed red curve with round markers) in the case of the IFOSMONDI-JFM method applied on the Battery Pack Cooling system

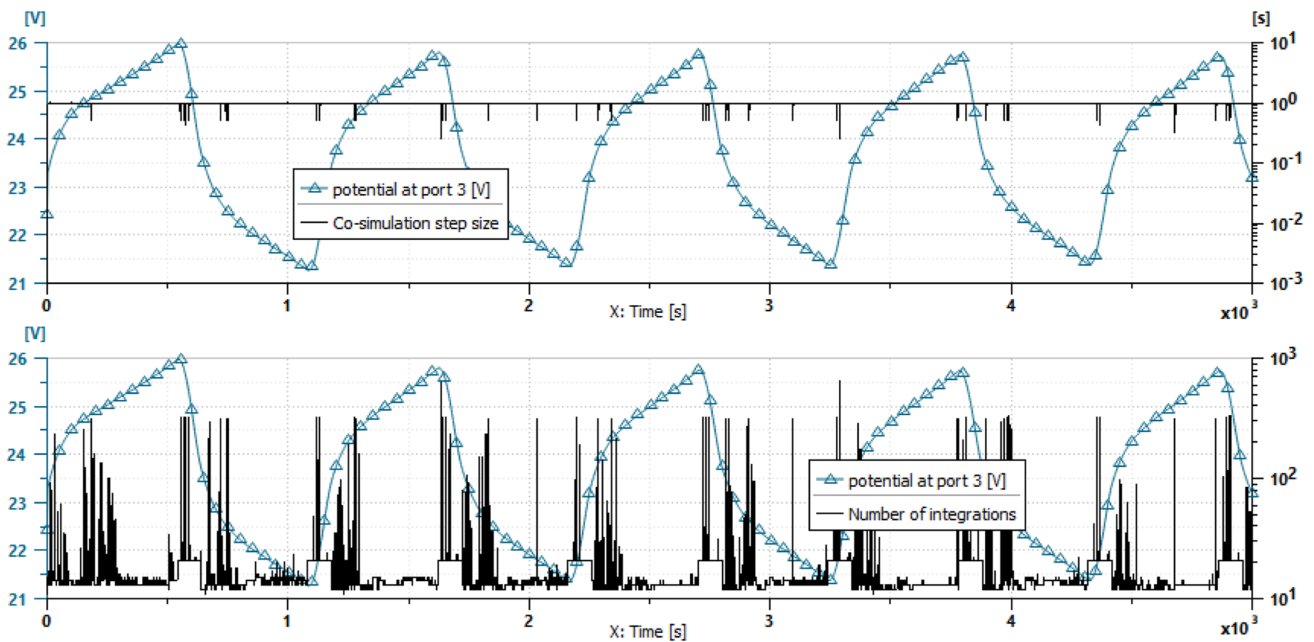

Fig. 24 Visualization of the connection between the co-simulation step size (upper straight curve, right y-scale), the number of integrations (lower straight curve, right y-scale) and a representative variable of interest of the system (superimposed blue curve with triangle markers) in the case of the fixed-point IFOSMONDI method applied on the Battery Pack Cooling system

iterates, [1]), this JFM is chosen in IFOSMONDI-JFM. The results are obtained with several values or ε both with IFOSMONDI-JFM and fixed-point IFOSMONDI methods.

A strong knowledge of the model is not required with these methods, yet an idea of the order of magnitude of the

co-simulation step size always helps. For this reason, we used the following parameters:

- $\delta t_{\text{min}} = 1$ ms, to be able to catch the fast interfaces dynamics, if any,
- $\delta t_{\text{max}} = 1$ s, to avoid missing events (peaks, slope changes, etc...), and
- $\delta t_{\text{init}} = 1$ ms for safety reasons (catch high dynamics at initialization).

In contrast, the explicit ZOH co-simulation method requires the step size to be chosen at the beginning, which implies that the user has a strong knowledge of the system. For this reason, we ran co-simulations with different values of the fixed co-simulation step size δt.

Please note that co-simulation with the explicit ZOH method ran with the same architecture than the IFOSMONDI methods. In other words, each co-simulation required 12 processes to run, regardless of the method. This by-design parallelism will therefore not bias the results below (Tables 3, 4).

Please note that, in the case of fixed-point IFOSMONDI, one iteration corresponds to a single integration (Table 5).

Please note that, in the case of Explicit ZOH, one co-simulation step corresponds to a single integration.

On the trade-off graph on Fig. 20, the more a co-simulation is valuable, the more it is close to the bottom-left corner, meaning that the run is accurate and fast. Every method follows a well-known phenomenon: the more a co-simulation is accurate, the slower it is. Graphically, this means that point corresponding to a given method goes on the right on the x-axis when they go down on the y-axis.

Nonetheless, both IFOSMONDI methods' curves are lower than the Explicit ZOH's curve and more on the left. This can be interpreted in two equivalent ways:

- at equivalent accuracy as Explicit ZOH, the IFOSMONDI methods (fixed-point and JFM) are faster
- at an equivalent computational time as Explicit ZOH, the IFOSMONDI methods are more accurate.

In addition, the trade-off curve of IFOSMONDI-JFM is lower and more on the right than the trade-off curve of fixed-point IFOSMONDI. It means that, with the same ε convergence criterion, IFOSMONDI-JFM is more accuracy-oriented than the fixed-point IFOSMONDI method.

Let's focus on the $\varepsilon = 10^{-8}$ cases. The average step size was 0.831 s for IFOSMONDI-JFM and 0.986 s for fixed-point IFOSMONDI, so let's compare the results with the run with the explicit ZOH method with a fixed co-simulation step size of 1 s. Two variables of interest are plotted on Fig. 21. To see the differences of accuracy between the runs, a focus on $t \in [3730, 3830]$ is presented on Fig. 22. On the latter, we can clearly see that both IFOSMONDI methods visually match the monolithic reference solution whereas the explicit ZOH has a delay and an overshoot.

Finally, the step size adaptation with the rule described in 3.4 can be visualize in both IFOSMONDI fixed-point and JFM methods with $\varepsilon = 10^{-8}$ together with the number of integrations (including the rejected steps) and one of the variable of interests of the system. Figures 23 and 24 show that the methods focus on the one hand on the stiff parts of the simulation by integrating more time and reducing the co-simulation step size, and on the other hand they save time on the non-stiff parts by increasing the co-simulation step size and iterating a smaller amount of time. This phenomenon can be explained by the fact that non-stiff models (or non-stiff parts of a simulation of models with variable stiffness) produce a version of the coupling constraint that is easier to satisfy (at a given ε tolerance) than stiff models (or stiff parts of a simulation of models with variable stiffness).

6 Conclusion

The IFOSMONDI-JFM algorithm takes advantages from the C^1 smoothness of fixed-point IFOSMONDI algorithm [6] without the delay that this smoothness implies in [5] (thanks to its iterative aspect), the coupling constraint is satisfied both at left and right of every communication time thanks to the underlying non-linear solvers of PETSc [2]. The iterative part does not need a finite differences estimation of the jacobian matrix like in [14] or a reconstruction of it like in [15].

The resulting algorithm even solves co-simulation problems for which the fixed-point formulation would involve a non-contractant coupling function $\boldsymbol{\Psi}_\tau$.

Thanks to its algebraic loop, the test case introduced in 5.1.1 shows this robustness as its difficulty can easily be increased or decreased in a quantifiable way. It can be a good candidate to benchmark the robustness of various co-simulation methods.

On the test-cases considered in this paper, the IFOSMONDI-JFM method either requires less iterations to converge when the parameterization enables both methods to solve the problem, or has a better accuracy than the one obtained with the fixed-point IFOSMONDI method. In the end, the time/accuracy trade off is similar for both methods.

The matrix-free aspect of the underlying solvers used with IFOSMONDI-JFM and their fast convergence are two of the causes of the small amount of integrations per step.

As the contractance of the fixed-point function is not always possible to analyze in the case of industrial cases in practise, one cannot know in advance if the fixed-point IFOSMONDI method will work or fail. The robustness of the newly introduced IFOSMONDI-JFM method brings a solution to this problem.

Table 6 Parameters of the NewtonLS method

PETSc argument: −snes_linesearch_ < ... >	Description	Value
type	Select line search type	bt
order	Selects the order of the line search for bt	3
norms	Turns on/off computation of the norms for basic line search	TRUE
alpha	Sets alpha used in determining if reduction in function norm is sufficient	0.0001
maxstep	Sets the maximum stepsize the line search will use	10^8
minlambda	Sets the minimum lambda the line search will tolerate	10^{-12}
damping	Damping factor used for basic line search	1
rtol	Relative tolerance for iterative line search	10^{-8}
atol	Absolute tolerance for iterative line search	10^{-15}
ltol	Change in lambda tolerance for iterative line search	10^{-8}
max_it	Maximum iterations for iterative line searches	40
keeplambda	Use previous lambda as damping	FALSE
precheck_picard	Use a correction that sometimes improves convergence of Picard iteration	FALSE

Table 7 Parameters of the Anderson method

PETSc argument: −snes_anderson_ < ... >	Description	Value
m	Number of stored previous solutions and residuals	30
beta	Anderson mixing parameter	1
restart_type	Type of restart	NONE
restart_it	Number of iterations of restart conditions before restart	2
restart	Number of iterations before periodic restart	30

Table 8 Parameters of the Ngmres method (not Ngmres with line search)

PETSc argument: −snes_ngmres_ < ... >	Description	Value
select_type	Choose the select between candidate and combined solution	DIFFERENCE
restart_type	Choose the restart conditions	DIFFERENCE
candidate	Use NGMRES variant which combines candidate solutions instead of actual solutions	FALSE
approxfunc	Linearly approximate the function	FALSE
m	Number of stored previous solutions and residuals	30
restart_it	Number of iterations the restart conditions hold before restart	2
gammaA	Residual tolerance for solution select between the candidate and combination	2
gammaC	Residual tolerance for restart	2
epsilonB	Difference tolerance between subsequent solutions triggering restart	0.1
deltaB	Difference tolerance between residuals triggering restart	0.9
single_reduction	Aggregate reductions	FALSE
restart_fm_rise	Restart on residual rise from x_M step	FALSE

Appendix A Parameters of the PETSc non-linear solvers

The JFMs mentioned in this document (see definition in Sect. 2.1) refer to PETSc non-linear solvers, so-called 'SNES' in the PETSc framework.

The parameters of these methods where the default one, except the explicitely mentioned ones. For the sake of reproducibility, the following tables recaps these options (Tables 6, 7, 8, 9). For further definition of their meaning, see [1, 2, 10].

Table 9 Parameters of the Ngmres with linsearch method

PETSc argument: −snes_ngmres_ < ... >	Description	Value
select_type	Choose the select between candidate and combined solution	LINESEARCH
⋮		
All other options of table 8 are the same		
⋮		
PETSc argument: −snes_linesearch_ < ... >	**Description**	**Value**
type	Select line search type	basic
order	Selects the order of the line search for bt	0
norms	Turns on/off computation of the norms for basic linesearch	TRUE
maxstep	Sets the maximum stepsize the line search will use	10^{8}
minlambda	Sets the minimum lambda the line search will tolerate	10^{-12}
damping	Damping factor used for basic line search	1
rtol	Relative tolerance for iterative line search	10^{-8}
atol	Absolute tolerance for iterative line search	10^{-15}
ltol	Change in lambda tolerance for iterative line search	10^{-8}
max_it	Maximum iterations for iterative line searches	1
keeplambda	Use previous lambda as damping	FALSE
precheck_picard	Use a correction that sometimes improves convergence of Picard iteration	FALSE

References

1. Anderson DGM (1965) Iterative procedures for nonlinear integral equations. J ACM 12:547–560
2. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Dalcin L, Dener A, Eijkhout V, Gropp WD, Karpeyev D, Kaushik D, Knepley MG, May DA, McInnes LC, Mills R.T, Munson T, Rupp K, Sanan P, Smith BF, Zampini S, Zhang H, Zhang H (2019) PETSc Web page. https://www.mcs.anl.gov/petsc
3. Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhäuser Press, pp 163–202
4. Benedikt M, Watzenig D, Zehetner J, Hofer A (2013) NEPCE—A nearly energy-preserving coupling element for weak-coupled problems and co-simulation. In: Proceedings of the International Conference on Computational Methods for Coupled Problems in Science and Engineering. pp 1–12
5. Busch M (2017) Performance improvement of explicit co-simulation methods through continuous extrapolation. In: IUTAM Symposium on solver-coupling and co-simulation. IUTAM Bookseries, vol 35, pp 57–80. IUTAM (2019). https://doi.org/10.1007/978-3-030-14883-6_4, iUTAM Symposium on Solver-Coupling and Co-Simulation, Darmstadt, Germany, September 18–20
6. Éguillon Y, Lacabanne B, Tromeur-Dervout D (2019) IFOSMONDI: a generic co-simulation approach combining iterative methods for coupling constraints and polynomial interpolation for interfaces smoothness. In: 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. pp 176–186. SCITEPRESS - Science and Technology Publications, Prague, Czech Republic, https://doi.org/10.5220/0007977701760186
7. Gomes C, Thule C, Broman D, Larsen PG, Vangheluwe H (2018) Co-simulation: a survey. ACM Comput Surv (CSUR) 51(3):1–33
8. Gu B, Asada HH (2004) Co-simulation of algebraically coupled dynamic subsystems without disclosure of proprietary subsystem models. J Dyn Syst Meas Contr 126(1):1–13. https://doi.org/10.1115/1.1648307
9. Kübler R, Schiehlen W (2000) Two methods of simulator coupling. Math Comput Model Dyn Syst 6(2):93–113. https://doi.org/10.1076/1387-3954(200006)6:2;1-M.FT093
10. Oosterlee CW, Washio T (2000) Krylov subspace acceleration of nonlinear multigrid with application to recirculating flows. SIAM J Sci Comput 21(5):1670–1690. https://doi.org/10.1137/S1064827598338093
11. Sadjina S, Pedersen E (2020) Energy conservation and coupling error reduction in non-iterative co-simulations. Eng Comput 36:1579–1587
12. Sadjina S, Kyllingstad LT, Skjong S, Pedersen E (2017) Energy conservation and power bonds in co-simulations: non-iterative adaptive step size control and error estimation. Eng Comput 33(3):607–620
13. Schierz T, Arnold M, Clauß C (2012) Co-simulation with communication step size control in an FMI compatible master algorithm. pp 205–214. https://doi.org/10.3384/ecp12076205
14. Schweizer B, Lu D (2015) Predictor/corrector co-simulation approaches for solver coupling with algebraic constraints. ZAMM Zeitschrift fur Angewandte Mathematik und Mechanik 95(9):911–938. https://doi.org/10.1002/zamm.201300191
15. Sicklinger S, Belsky V, Engelman B, Elmqvist H, Olsson H, Wüchner R, Bletzinger KU (2014) Interface Jacobian-based co-simulation. Ph.D. thesis. 10.1002/nme
16. Viel A (2014) Implementing stabilized co-simulation of strongly coupled systems using the functional mock-up interface 2.0. In: Proceedings of the 10 th International Modelica Conference. pp 213–223. Linköping University Electronic Press. March 10 – March 12, Lund, Sweden

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.