# Context-based polynomial extrapolation and slackened synchronization for fast multi-core simulation using FMI

Abir Ben Khaled[1] Laurent Duval[1] Mongi Ben Gaid[1] Daniel Simon[2]

[1]IFP Energies nouvelles, 1 et 4 avenue de Bois-Préau, 92852 Rueil-Malmaison, France
[2]INRIA and LIRMM - DEMAR team, 95 rue de la Galéra, 34090 Montpellier, France
abir.ben-khaled@ifpen.fr laurent.duval@ifpen.fr
mongi.ben-gaid@ifpen.fr daniel.simon@inria.fr

## Abstract

The growing complexity of systems, together with increasing available parallelism provided by multi-core chips, calls for the parallelization of simulation. Simulation speed-ups are expected from co-simulation and parallelization based on models splitting into loosely coupled sub-systems in the framework of Functional Mockup Interface (FMI). However, slackened synchronization between the sub-models and associated solvers running in parallel introduces integration errors, which must be kept inside predefined bounds. In this paper, context-based extrapolation is investigated to improve the trade-off between integration speed-ups, needing large communication steps, and simulation precision, needing frequent updates for the models inputs. An internal combustion engine, based on FMI for model exchange, is used to assess the parallelization methodology.

*Keywords: FMI; parallel simulation; signal processing; polynomial extrapolation; real-time; context-based decision*

## 1 Introduction

During the design process of complex systems, such as in automotive, simulation is proven to be an indisputable step between concept design and prototype validation. Realistic simulations allow for the preliminary evaluation, tuning and possibly redesign of proposed solutions ahead of implementation, thus lowering the risks. To be confident in the result, building such simulations requires high-fidelity models both for the components and for their interaction.

Currently, building high-fidelity system-level models of cyber-physical systems in general and automotive cars in particular, is a challenging duty. One problem is the diversity of models, designed for different environments, provided by various multi-disciplinary teams. Distinctive environments are preferred for a specific use due to specific strengths (modeling language, libraries, solvers, cost...). The FMI specification has been proposed to improve this issue [1].

However, the simulation of high-fidelity models is time consuming, and reaching real-time constraints is out of the capabilities of single-threaded simulations running on single cores. Simulation speed-ups are needed, in particular by splitting the systems into sub-models to be executed in parallel on currently available multi-core chips.

Unfortunately most of the existing simulation software are currently unable to exploit multi-core plateforms, as they rely on sequential Ordinary Differential Equations (ODE) and Differencial Algebraic Equations (DAE) solvers. The co-simulation approaches can provide significant improvements by allowing to simulate together models coming from different areas, and to validate both the individual behaviors and their interaction [2]. The simulators may be exported from original authoring tools as Functional Mock-up Units (FMUs), and then imported in a co-simulation environment. Hence, they cooperate at run-time, thanks to the FMI definitions of their interfaces, and to the master algorithms of these environments.

The "FMI for model exchange" framework allows for solving independently the sub-models using custom solvers. In this context, several methods have been already proposed to perform real-time distributed simulation of complex physical models. For example, in [3], the study focused on the case of fixed-step solvers. Then, in [4], the study was extended to handle the case of variable time-step solvers.

However, accounting for the dependencies between

the sub-models needs to synchronize them at some time intervals. Certainly, this synchronization avoids the propagation of numerical errors in the simulation results and guarantees their correctness. Unfortunately, these synchronization constraints also lead to waiting periods and idle time of some processors. Consequently it decreases the potential efficiency of the threaded parallelism existing in multi-core platforms.

To overcome this limitation, and to more efficiently exploit the available parallelism, the dependencies constraints should be relaxed as far as possible while keeping accumulated errors under control. In a first step, this can be performed by a well done system decomposition that minimizes the dependencies between the sub-models. For example, a method was proposed in [5] for distributed simulation using a technology based on bilateral delay lines called transmission line modeling (TLM), where the decoupling point is chosen when the variables change slowly and the time-step of the solver is relatively small.

Unfortunately, most often perfect decoupling cannot be reached and data dependencies still exist between parallel blocks. Some synchronization between them must be kept tight through small communication steps between models, which prevents the variable time-step solvers to reach large integration steps.

It is proposed that the synchronization steps can be stretched out with limited deterioration of the simulation precision, thanks to a well-suited, albeit simple, context-based polynomial extrapolation of the exchanged data beyond the synchronization points between sub-models.

This paper is organized as follows. First, a formal model of a hybrid dynamical system is given and a model of the integration errors due to slack synchronization is sketched in Section 2. The background on prediction and polynomial prediction algorithms are developed in Section 3. The principles for context-based extrapolation, to cope with the hybrid nature of the models, are exposed in Section 4. The methodology is assessed in Section 5 using the model of an internal combustion engine.

# 2 Motivation for extrapolation

## 2.1 Model formalization

Consider a hybrid dynamical system $\Sigma$ described by a set of nonlinear differential equations:

$$\begin{aligned} \dot{\boldsymbol{X}} &= \boldsymbol{f}(t,\boldsymbol{X},\boldsymbol{D},\boldsymbol{U}) \quad \text{for} \quad t_n \le t < t_{n+1}, \\ \boldsymbol{Y} &= \boldsymbol{g}(t,\boldsymbol{X},\boldsymbol{D},\boldsymbol{U}), \end{aligned}$$

where $\boldsymbol{X} \in \mathbb{R}^{n_X}$ is the continuous state vector, $\boldsymbol{D} \in \mathbb{R}^{n_D}$ is the discrete state vector, $\boldsymbol{U} \in \mathbb{R}^{n_U}$ is the input vector, $\boldsymbol{Y} \in \mathbb{R}^{n_Y}$ is the output vector and $t \in \mathbb{R}^+$ is the time.

The sequence $(t_n)_{n\ge 0}$ of strictly increasing time instants represents discontinuity points called "state events", which are the roots of the equation

$$\boldsymbol{h}(t,\boldsymbol{X},\boldsymbol{D},\boldsymbol{U}) = 0.$$

The function $\boldsymbol{h}$ is usually called zero-crossing function or event indicator. It is used for *event detection* and *location* [6].

At each time instant $t_n$, a new continuous state vector can be computed as a result of the *event handler*

$$\boldsymbol{X}(t_n) = \boldsymbol{I}(t_n,\boldsymbol{X},\boldsymbol{D},\boldsymbol{U}),$$

and a new discrete state vector can be computed as a result of discrete state update

$$\boldsymbol{D}(t_n) = \boldsymbol{J}(t_{n-1},\boldsymbol{X},\boldsymbol{D},\boldsymbol{U}).$$

If no discontinuity affects a component of $\boldsymbol{X}(t_n)$, the right limit of this component will be equal to its value at $t_n$.

It is assumed that $\Sigma$ is well posed in the sense that a unique solution exists for each admissible initial conditions $\boldsymbol{X}(t_0)$ and $\boldsymbol{D}(t_0)$ and that consequently $\boldsymbol{X}$, $\boldsymbol{D}$, $\boldsymbol{U}$, and $\boldsymbol{Y}$ are piece-wise continuous functions in $[t_n, t_{n+1}]$.

To execute the system in parallel, the model must be split into several sub-models. Assume for simplicity, that the system is decomposed into two subsystems as in Figure 1. Our approach generalizes to any decomposition into $N$ blocks of system $\Sigma$.

Therefore, the system can be written as:

$$\begin{cases} \dot{\boldsymbol{X}}_1 = \boldsymbol{f}_1(\boldsymbol{X}_1,\boldsymbol{X}_2,\boldsymbol{D}_1,\boldsymbol{U}_1) \\ \boldsymbol{Y}_1 = \boldsymbol{g}_1(\boldsymbol{X}_1,\boldsymbol{X}_2,\boldsymbol{D}_1,\boldsymbol{U}_1) \end{cases} \text{and} \begin{cases} \dot{\boldsymbol{X}}_2 = \boldsymbol{f}_2(\boldsymbol{X}_1,\boldsymbol{X}_2,\boldsymbol{D}_2,\boldsymbol{U}_2) \\ \boldsymbol{Y}_2 = \boldsymbol{g}_2(\boldsymbol{X}_1,\boldsymbol{X}_2,\boldsymbol{D}_2,\boldsymbol{U}_2) \end{cases}$$

with $\boldsymbol{X} = [\boldsymbol{X}_1\ \boldsymbol{X}_2]^T$ and $\boldsymbol{D} = [\boldsymbol{D}_1\ \boldsymbol{D}_2]^T$, where $^T$ denotes the matrix transpose.

Here, $\boldsymbol{U}_1$ are the inputs needed for $\Sigma_1$ and $\boldsymbol{U}_2$ are the inputs needed for $\Sigma_2$. In other words, $\boldsymbol{U}_1 \cup \boldsymbol{U}_2 = \boldsymbol{U}$


DOI 10.3384/ECP14096225

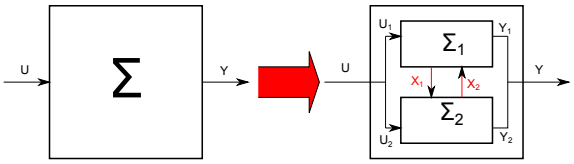


Figure 1: System splitting for parallelization.

and $\boldsymbol{U}_1 \cap \boldsymbol{U}_2$ can be an empty set or not according to the achieved decoupling.

In the same way, $\boldsymbol{Y}_1$ are the outputs produced by $\Sigma_1$ and $\boldsymbol{Y}_2$ are the outputs produced by $\Sigma_2$. In other words, $\boldsymbol{Y}_1 \cup \boldsymbol{Y}_2 = \boldsymbol{Y}$ and $\boldsymbol{Y}_1 \cap \boldsymbol{Y}_2 = \emptyset$.

To perform the numerical integration of the whole multivariable system, each of these simulators needs to exchange, at communication points, the data needed by the others (see Figure 2). To speed up the integration, the parallel branches must be as independent as possible, so that they are synchronized at a rate $P$ by far slower than their internal integration step $h_n$ ($P \gg h_n$). Therefore, between communication points, each simulator integrates at its own rate (assuming a variable step solver), and considers that the data incoming from others simulators is hold as constant.

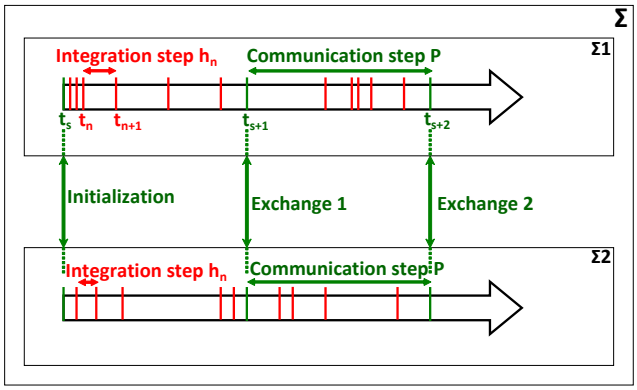


Figure 2: $\Sigma$ split into $\Sigma_1$ and $\Sigma_2$ for parallel simulation.

It is likely that large communication intervals allow to speed up the numerical integration, but may result in integration errors and poor confidence in the final result. Modeling the errors induced by slack synchronization is a first step to find effective directions to improve the trade-offs between integration speed and accuracy.

## 2.2 Integration errors and parallelism

To compute the next state value $\boldsymbol{X}_i(t_{n+1}), i = 1,2$ (see Figure 3), the numerical solver needs at least the values of $\boldsymbol{X}_i(t_n)$ and $\dot{\boldsymbol{X}}_i(t_n) = \boldsymbol{f}_i(\boldsymbol{X}(t_n))$ (e.g. for Euler integration). The inputs and discrete states are omitted for clarity.

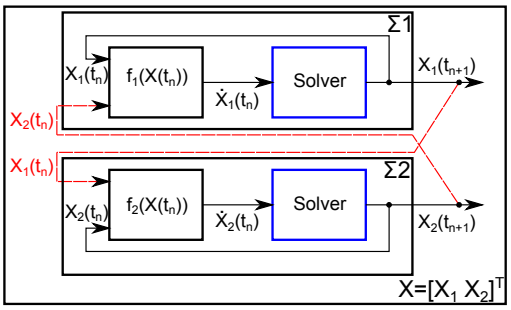


Figure 3: System's internal composition.

When computing $\dot{\boldsymbol{X}}_1(t_n) = \boldsymbol{f}_1(\boldsymbol{X}(t_n))$, the value of the local variable $\boldsymbol{X}_1(t_n)$ is always available. This is not the case for $\boldsymbol{X}_2(t_n)$, which is computed in a parallel branch. In fact, $\boldsymbol{X}_2$ is only available in branch 1 at synchronization with interval $P$, which is larger than the integration step $h_n$. In other words, $\boldsymbol{X}_2(t_n)$ is available only when the time $t_n$ corresponds with a synchronization point $t_s$ (see Figure 2), otherwise its estimated value is the one transmitted at the previous synchronization point. Let us evaluate the evolution of integration errors due to slack synchronization between the parallel branches when computing $\dot{\boldsymbol{X}}_1(t_n) = \boldsymbol{f}_1(\boldsymbol{X}(t_n))$. The analysis on $\Sigma_1$ remains valid for $\Sigma_2$.

The influence of using a delayed value of $\boldsymbol{X}_2$ in $\boldsymbol{f}_1(.)$ (respectively $\boldsymbol{X}_1$ in $\boldsymbol{f}_2(.)$) is due to the lack of updated data during a delay $\tau$, represented by the difference between the current integration time $t_n$ and the last synchronization time $t_s$ as

$$\tau = t_n - t_s \tag{1}$$

with

$$t_s = P \left\lfloor \frac{t_n}{P} \right\rfloor$$

therefore

$$t_s = \begin{cases} lP & \text{when} \quad t_n = l.P \quad l \in \mathbb{N}^* \\ (l-1)P & \text{when} \quad t_n < l.P \quad l \in \mathbb{N}^* \end{cases}$$

leading to

$$\begin{cases} \tau = 0 & \text{when} \quad t_n = t_s \\ \tau > 0 & \text{when} \quad t_n > t_s \end{cases}$$

Therefore, the induced error at $t_{n+1}$ in the subsystem $\Sigma_1$, denoted $\boldsymbol{E}_1(t_{n+1})$, is the difference between $\boldsymbol{X}_1(t_{n+1})$ for the unsplit model (2) and $\tilde{\boldsymbol{X}}_1(t_{n+1})$ for the split model (3):

$$\boldsymbol{X}_1(t_{k+1}) = \boldsymbol{X}_1(t_k) + h_k \boldsymbol{f}_1(\boldsymbol{X}_1(t_k), \boldsymbol{X}_2(t_k)), \;\; k \in \{0, \ldots, n\} \tag{2}$$

$$\tilde{\boldsymbol{X}}_1(t_{k+1}) = \begin{cases} \boldsymbol{X}_1(t_{k+1}) & k = 0 \\ \tilde{\boldsymbol{X}}_1(t_k) + h_k \boldsymbol{f}_1(\tilde{\boldsymbol{X}}_1(t_k), \tilde{\boldsymbol{X}}_2(t_k - \tau)) & k \geq 1 \end{cases} \tag{3}$$

In other words,

$$\begin{aligned} \boldsymbol{E}_1(t_{n+1}) &= \sum_{k=0}^{n} \boldsymbol{E}_1(t_k) \\ &\quad + h_n[\boldsymbol{f}_1(\boldsymbol{X}_1(t_n), \boldsymbol{X}_2(t_n)) - \boldsymbol{f}_1(\tilde{\boldsymbol{X}}_1(t_n), \tilde{\boldsymbol{X}}_2(t_n - \tau))] \\ &= \boldsymbol{E}_{1,p}(t_n) + \boldsymbol{E}_{1,c}(t_{n+1}) \end{aligned}$$

where

$$\begin{aligned} \boldsymbol{E}_{1,c}(t_{n+1}) &= h_n[\boldsymbol{f}_1(\boldsymbol{X}_1(t_n), \boldsymbol{X}_2(t_n)) - \boldsymbol{f}_1(\tilde{\boldsymbol{X}}_1(t_n), \tilde{\boldsymbol{X}}_2(t_n - \tau))] \\ \boldsymbol{E}_{1,p}(t_n) &= \sum_{k=0}^{n} \boldsymbol{E}_1(t_k) \end{aligned} \tag{4}$$

Here $\boldsymbol{E}_{1,c}(t_{n+1})$ is the current error generated at $t_{n+1}$ whatever a synchronization or not. So, the global decoupling error $\boldsymbol{E}_1(t_{n+1})$ is the result of the accumulation of past errors $\boldsymbol{E}_{1,p}(t_n)$ and the current error $\boldsymbol{E}_{1,c}(t_{n+1})$. As a conclusion, to achieve a correct result, two conditions must be met for the current (local) error and the global error:

- $|\boldsymbol{E}_{1,c}(t_{n+1})| < \epsilon_{\text{loc}}$: allowed local error
- $|\boldsymbol{E}_1(t_{n+1})| < \epsilon_{\text{glo}}$: allowed global error

These conditions can be satisfied by acting on some parameters. Indeed, in (4), the delay error depends on the integration steps $h_n$ and on the delay $\tau$. The integration step $h_n$ is already adapted following the numerical solver strategy and the user-defined solver tolerance. The delay $\tau$, however, depends on the last synchronization time $t_s$, which is function of the synchronization period $P$.

The delay induced error tends to zero when the delay $\tau$ tends to zero, which means that the delay error can be eliminated with the synchronization interval set equal to the integration steps. In other words, all the parallel subsystems should be integrated at the same adaptive rate (in the case of adaptive synchronization period), or with same fixed time-step. These two assumptions are very restrictive, as they force to choose a global adequate time-step regardless the discontinuities and the stiffness of the sub-systems. Compared with the single-threaded simulation, the only possible speed-ups during a parallel execution would be brought by the brute force computation power of the multicore machine, reduced by the parallelization cost.

Therefore, considering a split model and a parallel execution, a trade-off must be found between acceptable simulation errors, thanks to tight enough synchronization, and simulation speed-ups thanks to decoupling between sub-models.

To add a degree of freedom to this trade-off achievement, we propose to extrapolate model inputs to compensate the stretching out of the communication steps between sub-models. Note that in this first approach of the polynomial extrapolation, the synchronization interval is considered as constant. Future enhancements will consider communication step size control, for which the error analysis and estimation can be inspired by [7]. Extrapolation is sensitive for different reasons:

- prediction should be efficient: causal, sufficiently fast and reliable;
- there exist no universal prediction scheme, efficient with every signal;
- polynomial prediction may fail in stiff cases [8] (cf. Section 4 for details).

We choose to base our extrapolation on polynomial prediction, which allows fast and causal calculations. The rationale is that, in this situation, the computing cost of a low-order polynomial predictor would be by far smaller than the extra model computations needed by shorten communication steps. Since such predictions would be accurate neither for any signal (for instance, blocky versus smooth signals) nor any signal behavior (slow variations versus steep onsets), we borrow a context-based approach , common with lossless image coders [9], such as GIF or PNG formats. The general aim of these image coders is to predict a pixel value based on a pattern of causal neighboring pixels. Compression is obtained when the prediction residues possess smaller intensity values, and more generally a better distribution (concentrated around close-to-zero values) than the pixels in the original image. They may thus be coded on smaller "bytes", using entropy coding techniques. In images, one distinguishes basic "objects" such as smooth-intensity varying regions, or edges with different orientations. Based on simple calculation of the prediction pattern pixels, different contexts are inferred (e.g. flat, smooth, $+45^o$ or $-45^o$ edges, etc.). Look-up table predictors are then used, depending on the context.

In the proposed approach, we build a heuristic table of contexts (in Section 4) based on a short frame of past samples, and affect a pre-determined polynomial


DOI 10.3384/ECP14096225

predictor to obtain a context-dependent extrapolated value. We now review the principles of extrapolation.

# 3 Causal polynomial prediction

## 3.1 Background on prediction

This section is dedicated to a peculiar instance of discrete time series, or signal, forecasting. The neighboring topics of prediction or extrapolation represent a large body of knowledge in signal processing [10], econometrics [11] or control [12].

In the present case, we consider a real-valued, regularly sampled signal $u$, with period $P$, known at synchronization or communication intervals. Prediction in general assumes the knowledge of signal formation models. Since very little is assumed on the signal's dynamics (no behavioral/explicit model is available, periodicity and regularity are unknown), and as we operate under real-time conditions, implying strong causality, only a tiny fraction of time series methods are practically applicable. Zeroth-order hold or nearest-neighbor extrapolation is probably the most natural, the less hypothetical, and the less computationally expensive forecasting method. It consists in using the latest known sample as the predicted value. It possesses small (cumulative) errors when the time series is relatively flat or its sampling rate is sufficiently high, with respect to the signal's dynamics. In other words, it is efficient when the time series is sampled fast enough to ensure small variations between two consecutive sampling times. However, it indirectly leads to undersampling related disturbances, that affect the signal content. They appear as quantization-like noise, offset or peak flattening.

In our co-simulation framework, communication intervals are not chosen arbitrarily small for computational efficiency. Thus, the slow variation of inputs and outputs cannot be ensured in practice. Hence, borrowing additional samples from the past known data and using higher-order extrapolation methods could be beneficial, provided a trade-off of cost and error is met. Different forecast methods of various fidelity and complexity may be efficiently evaluated. We focus here on polynomial methods, for their simplicity and ease of implementation, following initial works in [13, Chapter 16].

## 3.2 Notations

We denote by $P_{(\delta,\lambda)}$ the least-squares polynomial predictor of degree $\delta \in \mathbb{N}$ and prediction length $\lambda \in \mathbb{N}^*$. The prediction length $\lambda$ represents the number of past samples required for each prediction, performed in the least-squares sense [14, p. 227 sq.]. For convenience, we use a 0-last-sample-index convention: we re-index the frame of the $\lambda$ past samples such that the last known sample is indexed by 0. Computations for the prediction at relative time $\tau$ (loosely denoted by $u(\tau)$), defined in (1), thus require past samples $\{u_{1-\lambda}, u_{2-\lambda}, \ldots, u_0\}$. We first recall principles and formulas for a standard least-squares, degree-two or parabolic prediction. The general equations are derived next.

## 3.3 Polynomial prediction of degree $\delta = 2$

We look for the best fitting parabola, i.e. with degree $\delta = 2$, $u(t) = a_\delta + a_{\delta-1}t + a_{\delta-2}t^2$ to approximate the set of discrete samples $\{u_{1-\lambda}, u_{2-\lambda}, \ldots, u_0\}$. The prediction polynomial $P_{(2,\lambda)}$ is defined by the vector of polynomial coefficients $\boldsymbol{a} = [a_2, a_1, a_0]^T$. They are determined, in the least-squares sense [15], by minimizing the squared or quadratic, error:

$$e(\boldsymbol{a}) = \sum_{l=1-\lambda}^{0} \left(u_l - (a_2 + a_1 l + a_0 l^2)\right)^2 .$$

Note that the $l$ indices here are non-positive, between $1-\lambda$ and 0. The minimum error is obtained by solving the following system of equations (zeroing the derivatives with respect to each of the free variables $a_i$):

$$\forall i \in \{0,1,2\}, \quad \frac{\partial e(\boldsymbol{a})}{\partial a_i} = 0$$

namely:

$$\begin{cases} \sum_{l=1-\lambda}^{0} l^0 \left(u_l - (a_2 l^0 + a_1 l^1 + a_0 l^2)\right) = 0, \\ \sum_{l=1-\lambda}^{0} l^1 \left(u_l - (a_2 l^0 + a_1 i^l + a_0 l^2)\right) = 0, \\ \sum_{l=1-\lambda}^{0} l^2 \left(u_l - (a_2 l^0 + a_1 l^1 + a_0 l^2)\right) = 0. \end{cases} \tag{5}$$

The system in (5) may be rewritten as:

$$\begin{cases} \sum_{1-\lambda}^{0} u_l = a_2 \sum_{1-\lambda}^{0} l^0 + a_1 \sum_{1-\lambda}^{0} l^1 + a_0 \sum_{1-\lambda}^{0} l^2 , \\ \sum_{1-\lambda}^{0} l u_l = a_2 \sum_{1-\lambda}^{0} l^1 + a_1 \sum_{1-\lambda}^{0} l^2 + a_0 \sum_{1-\lambda}^{0} l^3 , \\ \sum_{1-\lambda}^{0} l^2 u_l = a_2 \sum_{1-\lambda}^{0} l^2 + a_1 \sum_{1-\lambda}^{0} l^3 + a_0 \sum_{1-\lambda}^{0} l^4 . \end{cases}$$

Let $m^d = \sum_{l=0}^{\lambda-1} l^{\delta-d} u_{-l}$ (here the indices $l$ are positive) denote the $(\delta-d)$-th moment of the frame $u_i$, and $\boldsymbol{m}$ the vector of moments $[m^2, -m^1, m^0]^T$. We express the sums of integer powers by $\Sigma_\lambda^d = \sum_{i=0}^{\lambda-1} i^d$. Closed-form expressions exist for $\Sigma_\lambda^d$, involving Bernoulli sequences [16]. For instance:

- $\Sigma_\lambda^0 = \lambda$,
- $\Sigma_\lambda^1 = (\lambda-1)\lambda/2$,
- $\Sigma_\lambda^2 = (\lambda-1)\lambda(2\lambda-1)/6$,
- $\Sigma_\lambda^3 = (\lambda-1)^2\lambda^2/4$,
- $\Sigma_\lambda^4 = (\lambda-1)\lambda(2\lambda-1)(3\lambda^2-3\lambda-1)/30$.

We now form the matrix $\boldsymbol{Z}_{(2,\lambda)}$ of sums of powers (depending on $\delta = 2$ and $\lambda$):

$$\boldsymbol{Z}_{(2,\lambda)} = \begin{bmatrix} \Sigma_\lambda^0 & -\Sigma_\lambda^1 & \Sigma_\lambda^2 \\ -\Sigma_\lambda^1 & \Sigma_\lambda^2 & -\Sigma_\lambda^3 \\ \Sigma_\lambda^2 & -\Sigma_\lambda^3 & \Sigma_\lambda^4 \end{bmatrix}.$$

The system in (5) rewrites:

$$\begin{bmatrix} m^2 \\ -m^1 \\ m^0 \end{bmatrix} = \begin{bmatrix} \Sigma_\lambda^0 & -\Sigma_\lambda^1 & \Sigma_\lambda^2 \\ -\Sigma_\lambda^1 & \Sigma_\lambda^2 & -\Sigma_\lambda^3 \\ \Sigma_\lambda^2 & -\Sigma_\lambda^3 & \Sigma_\lambda^4 \end{bmatrix} \times \begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix}$$

or $\boldsymbol{m} = \boldsymbol{Z}_{(2,\lambda)} \times \boldsymbol{a}$. Now we want to find the value predicted by $P_{(2,\lambda)}$ at time $\tau$. Let $\boldsymbol{\tau}_2 = [1, \tau, \tau^2]^T$ be a vector of $\tau$ powers. Then $u(\tau)$ is equal to $a_2 + a_1\tau + a_0\tau^2 = \boldsymbol{\tau}_2^T \times \boldsymbol{a}$. Finally, $\boldsymbol{Z}_{(2,\lambda)}$ is always invertible, provided that $\lambda > \delta$. Its inverse is denoted $\boldsymbol{Z}_{(-2,\lambda)}$. It thus does not need to be updated in real-time. It may be computed off-line, numerically or even symbolically. Hence:

$$u(\tau) = \left(\boldsymbol{\tau}_2^T \times \boldsymbol{Z}_{(-2,\lambda)}\right) \times \boldsymbol{m}.$$

The vector $\boldsymbol{\tau}_2$ and $\boldsymbol{Z}_{(-2,\lambda)}$ are fixed, and the product $\boldsymbol{\tau}_2^T \times \boldsymbol{Z}_{(-2,\lambda)}$ may be stored at once. Thus, for each prediction, the only computations are the update of the vector $\boldsymbol{m}$ and his product with the aforementioned stored matrix. It thus enables look-up-table-based predictions, which helps to reduce propagation errors in matrix computations.

### 3.4 General formulas

Inferring from the previous example, we easily get a more generic extrapolation pattern in its matrix form:

$$u(\tau) = \begin{bmatrix} 1 & \tau & \cdots & \tau^\delta \end{bmatrix} \times \begin{bmatrix} \Sigma_\lambda^0 & -\Sigma_\lambda^1 & \cdots & (-1)^\delta\Sigma_\lambda^\delta \\ -\Sigma_\lambda^1 & \iddots & \iddots & \vdots \\ \vdots & \iddots & \iddots & \vdots \\ (-1)^\delta\Sigma_\lambda^\delta & \cdots & \cdots & \Sigma_\lambda^{2\delta} \end{bmatrix}^{-1} \times \begin{bmatrix} m^\delta \\ -m^{\delta-1} \\ \vdots \\ (-1)^\delta m^0 \end{bmatrix}.$$

Note $\boldsymbol{\tau}_\delta = [1, \tau, \cdots, \tau^\delta]^T$, then:

$$u(\tau) = \boldsymbol{\tau}_\delta^T \boldsymbol{Z}_{(-\delta,\lambda)} \boldsymbol{m}.$$

As in the previous case, only $\boldsymbol{m}$ and one matrix product need be computed in real-time. When $\delta = 0$, one easily sees that:

$$u(\tau) = \frac{m^0}{\Sigma_0^\lambda} = \frac{u_{1-\lambda} + \cdots + u_0}{\lambda},$$

that is, the running average of past frame values, reducing to the zeroth-order hold when $\lambda = 1$. Although the matrix formulation is convenient, actual computation does not require true matrix calculus, especially for small degrees $\delta$. For instance, $P_{(1,3)}$ yields the simple estimator form: $u(\tau) = \frac{\tau}{2}(u_0 - u_{-2}) + \frac{1}{6}(5u_0 + 2u_{-1} - u_{-2})$.

## 4 Context-based extrapolation

Actual complex systems usually present non-linearities and discontinuities, so that it is hard to predict their future behavior from past observations. Moreover the considered models are generated using the FMI for model exchange framework, which does not provide the inputs' derivatives (conversely with the FMI for co-simulation architecture). Hence the previously described polynomial prediction cannot correctly extrapolate along all the system trajectories.

For example, [17] studies a method based on a sequential implementation of continuous dynamical systems that uses a constant, linear or quadratic extrapolation and a linear interpolation to improve the accuracy of the modular time integration. The study shows that the method is successful for non-stiff systems and it


DOI 10.3384/ECP14096225

fails for the stiff case. The context-based extrapolation is then performed to account for steps, stiffness, discontinuities or weird behavior, and use adapted extrapolation to limit excessively wrong prediction.

Keeping with the previous 0-last-sample-index convention, and for the sake of simplicity, we first define a measure of variation based on the last three samples: $d_0 = u_0 - u_{-1}$ and $d_1 = u_{-1} - u_{-2}$, the last and previous differences. Their absolute values are compared with two thresholds, $\gamma_0$ and $\gamma_{-1}$, respectively. We then define three complementary conditions:

- $O$ if $|d_i| = 0$;
- $C_i$ if $0 < |d_i| \leq \gamma_i$;
- $\overline{C_i}$ if $|d_i| > \gamma_i$;

We can now define the six-context Table 1, and examples for their associated heuristic polynomial predictors. The six contexts form a partition, i.e. they are mutually exclusive, and cover all possible options for a hybrid dynamical system. They are illustrated in Figure 4. Their names represent their behavior. For instance, the flat context addresses steady signals, for which a mere zeroth-order hold suffices, hence $P_{(0,1)}$. The calm context represents a sufficiently sampled situation, where value increments over time remain below fixed thresholds. In this case, the signal is relatively regular, and could be approximated by a quadratic polynomial, for instance $P_{(2,5)}$. For the "flat" and "jump" contexts, there is an additional procedure which consists in resetting the extrapolation to prevent inaccurate prediction. For example, when context 1 is chosen just after context 5, the quadratic extrapolation $P_{(2,5)}$ requires 5 valid samples, whereas the last 3 only are relevant.

Our two-threshold is relatively simple. Hence, the choice of the thresholds $\gamma_0$ and $\gamma_{-1}$, is potentially crucial. For instance, fixed values may reveal inefficient under important amplitude or scale variation of signal. Hence, we have chosen here to compute them, in a running manner, on the past frame $\{u_{1-\omega}, \ldots, u_{-3}\}$. With excessively low thresholds, high-order extrapolations would be rarely chosen, loosing the benefits of predictions. Too high thresholds would in contrast suffer from any unexpected jump or noise. As the contexts are based on backward derivatives, we have used in the simulations presented here the mid-range statistical estimator of their absolute values. This amounts to set: $\gamma_0 = \gamma_{-1} = \frac{1}{2} \max_{i \in [1-\omega, \ldots, -3]} (|u_i - u_{i+1}|)$.

Table 1: Summary of the six-context Table.

| n(ame) | # | $\lvert d_{-2}\rvert$ | $\lvert d_{-1}\rvert$ | $d_{-2}.d_{-1}$ | $(\delta, \lambda)$ |
|---|---|---|---|---|---|
| f(lat) | 0 | $O$ | $O$ | $O$ | $(0,1)$ |
| c(alm) | 1 | $C_1$ | $C_2$ | any | $(2,5)$ |
| m(ove) | 2 | $C_1$ | $\overline{C_2}$ | any | $(0,1)$ |
| r(est) | 3 | $\overline{C_1}$ | $C_2$ | any | $(0,2)$ |
| t(ake) | 4 | $\overline{C_1}$ | $\overline{C_2}$ | $>0$ | $(1,3)$ |
| j(ump) | 5 | $\overline{C_1}$ | $\overline{C_2}$ | $<0$ | $(0,1)$ |

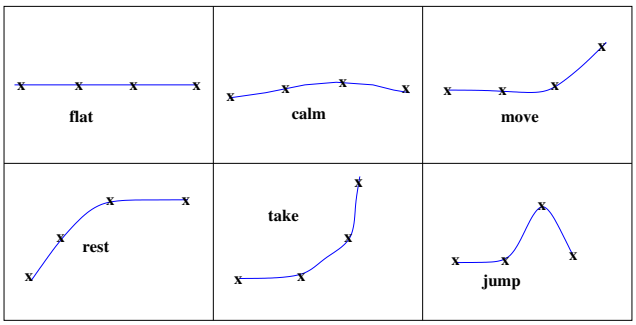


Figure 4: Illustration for context table in Table 1.

# 5 Case study

## 5.1 Engine simulator

In this study, a Spark Ignition (SI) RENAULT F4RT engine has been modeled. It is a four-cylinder in-line Port Fuel Injector (PFI) engine in which the engine displacement is 2000 $cm^3$. The air path (AP) consists in a turbocharger with a mono-scroll turbine controlled by a waste-gate, an intake throttle and a downstream-compressor heat exchanger.

The engine model was developed using the ModEngine library [18]. ModEngine is a Modelica [19] library that allows the modeling of a complete engine with diesel and gasoline combustion models. Requirements for the ModEngine library were derived from the existing IFP-Engine AMESim[1] library. ModEngine contains more than 250 sub-models. It has been developed to allow the simulation of a complete virtual engine using a time-scale related to fractions of the crankshaft angle. A variety of elements are available to build representative models for engine components, such as turbocharger, wastegate, gasoline or Diesel injectors, valve, air path, EGR loop etc. ModEngine is currently functional in the Dymola tool[2].

The engine model and the split parts were imported into xMOD model integration and virtual experimen-

[1] www.lmsintl.com/imagine-amesim-1-d-multi-domain-system-simulation

[2] www.3ds.com/products/catia/portfolio/dymola

tation tool [20], using the FMI export features of Dymola. The engine model has 118 state variables and 312 event indicators (of discontinuities).

### 5.2 Decomposition approach

The partitioning of the engine model is performed by separating the four-cylinder from the air path (AP), then by isolating the cylinders ($C_i$, for $i \in [1,2,3,4]$) from each other. This kind of splitting allows for the reduction of the number of events acting on each subsystem. In fact, the combustion phase raises most of the events, which are located in the firing cylinder. The solver can process them locally during the combustion cycle of the isolated cylinder, and then enlarge its integration time-step until the next cycle.

From a thermodynamic point of view, the cylinders are loosely coupled, but a mutual data exchange does still exist between them and the air path. The model is split into 5 components and governed by a basic controller denoted CTRL. It gathers 91 inputs and 98 outputs.

## 6 Tests and results

Tests are performed on a platform with 16 GB RAM and 2 "Intel Xeon" processors, each running 8 cores at 3.1 GHz.

### 6.1 Reference simulations

The model validation is based on the observation of some quantities of interest as the intake and exhaust manifold pressures, air-fuel equivalence ratio and torque. These outputs are computed using LSODAR which is a variable time-step solver with a root-finding capability to detect the events occurring during the simulation. It has also the ability to adapt the integration method depending on the observed system stiffness.

The simulation reference $Y_{\text{ref}}$ is built from the integration of the entire engine model, the solver tolerance (tol) being decreased until reaching stable results, which is reached for tol $= 10^{-7}$ (at the cost of an unacceptable slow simulation speed).

Then, to explore the trade-offs between the simulation speed and precision, simulations are run with increasing values of the solver tolerance until reaching a desired relative integration error $Er$, defined by (6)

$$Er(\%) = \frac{100}{N} \cdot \sum_{i=0}^{N-1} \left( \left| \frac{Y_{\text{ref}}(i) - Y(i)}{Y_{\text{ref}}(i)} \right| \right) \qquad (6)$$

with $N$ the number of saved points during 1 s of simulation. Iterative runs showed that the relative error converge to a desired error ($Er \leq 1\,\%$) for tol $= 10^{-4}$. The single thread simulation of the whole engine with LSODAR and tol $= 10^{-4}$ provides the simulation execution time reference, to which the parallel versions are compared. When using the split model, each of its 5 components is assigned to a dedicated core and integrated by LSODAR with tol $= 10^{-4}$.

### 6.2 Effect of the context-based extrapolation on accuracy

To explore the effect of extrapolation on accuracy, the communication step has been set to 250 µs in a first set of experiments. This value has been chosen to provide acceptable results for the accuracy ($Er \approx 1\,\%$), while being large enough to make extrapolation useful.

The tests show that performing only a fixed polynomial prediction (conventional first and second order extrapolation) on the engine model fails, with integration errors larger than for the reference simulation. This is due to the hybrid nature of the model, for which the extrapolation failures are caused by discontinuities, and also by sharp variations of some variables at specific instants. These cases totally waste the gain in precision due to successful extrapolation in the other parts of the state trajectories.

In contrast, using the context-based polynomial predictor, the outputs of the simulation are almost always closer to the reference trajectory than those computed when considering the inputs hold as constant (Figure 5).

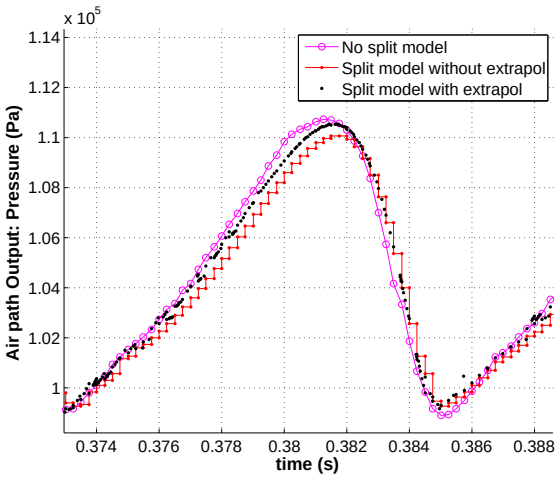


Figure 5: Airpath output: pressure.

Figure 6 shows that using context-based extrapola-


DOI 10.3384/ECP14096225

tion, the prediction step is discarded when there is a discontinuous behavior in the signal, and that the degree of the predictor is adapted according to the signal slope (Figure 6).

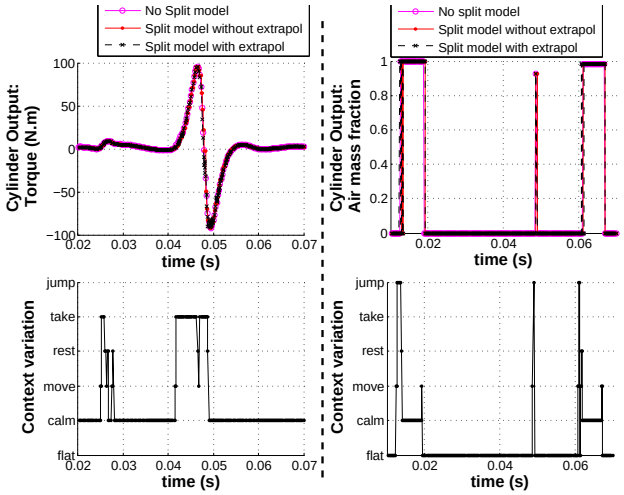


Figure 6: Context behavior during simulation.

The cumulative relative integration error on a long simulation run is computed in Table 2. It shows that the context-based extrapolation efficiently decreases this error for the chosen variables, for example by 63 % for the temperature and by 72.5 % for the fuel density.

Table 2: Relative integration error.

| Outputs | Er(%) w/o extrapolation | Er(%) w/ extrapolation |
|---|---|---|
| Pressure | 0.499 | 0.304 |
| Temperature | 0.511 | 0.19 |
| Air density | 0.784 | 0.31 |
| Fuel density | 3.55 | 0.978 |
| Burned gas density | 4.99 | 3.47 |

### 6.3 Effect of the context-based extrapolation on simulation time

The ultimate objective of extrapolation is to decrease the simulation by stretching out the synchronization interval, while keeping the relative integration error $Er$ inside predefined bounds. Indeed, widening the communication step from 100 µs to 250 µs without extrapolation (Figure 7) saves time but increases the error (e.g. 6.97 % for the burned gas density and 340.5 % for the fuel density).

Using the extrapolation for the 250 µs step fortunately decreases the relative error to values close to, or below, those measured for the 100 µs step with frozen inputs.

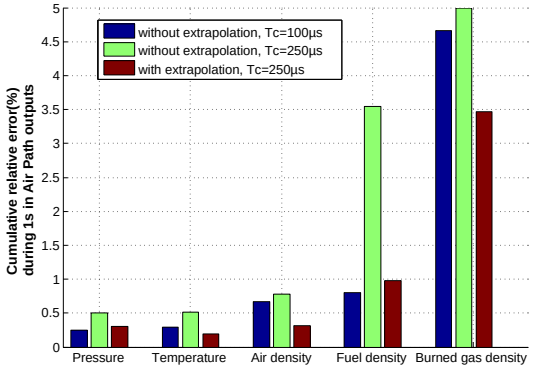


Figure 7: Cumulative relative error using different communication steps.

Table 3 shows the simulation speed-up compared with the single-threaded reference. First note that when splitting the model into 5 threads integrated in parallel on 5 cores, the speed-up is supra-linear w.r.t. the number of cores. Indeed, the containment of events detection and handling inside small subsystems allows for solvers accelerations, enough to over-compensate the multi-threading costs. Secondly, it appears that combining the enlarged communication step and the context-based extrapolation, the 10 % extra speed-up is reached without loss for the relative error. Even more surprising, using the extrapolation slightly speeds-up the simulation, possibly because the inputs shaped by the predictor enables a faster convergence of the solver step.

Table 3: Simulation speed-up.

| Communication time | 100 µs | 250 µs | |
|---|---|---|---|
| Extrapolation | No | No | Yes |
| Speed-up | 8.9 | 10.01 | 10.07 |

## 7 Conclusion and future work

The aim of this work is to speed up the numerical integration of hybrid dynamical systems, eventually until reaching a real-time execution, while keeping the integration errors inside controlled bounds. The basic approach consists in splitting the system into submodels, which are integrated in parallel. Using large synchronization intervals between the branches allows for numerical integration speed-ups. However, slack

synchronization intervals may generate integration errors in the final result.

In this paper, the errors caused by the slack synchronization are modeled, giving directions to find effective trade-offs between integration speed and accuracy. Then, an approach of stretching out the communication steps while keeping a predefined integration precision is proposed. Rather than using costly small integration and communication steps, it uses extrapolations of the behavior of the models over the synchronization intervals. Test results on a hybrid dynamical engine model, show that well chosen context-based extrapolation allows for an effective speed-up of the simulation with negligible computing overheads.

This work shows that properly-chosen context-based extrapolation, combined with model splitting and parallel integration, can potentially improve the speed/precision trade-off needed to reach real-time simulation. However, the accuracy could be widely improved by accessing on the input derivatives of the models. This is the case for the FMI for co-simulation, and it would be highly useful to also integrate this feature in the FMI for model exchange. Future works intend to improve the context-based extrapolation algorithm, to make it more subtly aware of data freshness and even more decrease the prediction induced integration errors. Another possibility is to process the input signals to separate them into simpler components, easier to predict with different predictors, and to cope with noise. When it comes to polynomials, wavelet pre-processors [21] could be useful, as they play an important role in polynomial model fitting.

# References


[1] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The functional mockup interface for tool independent exchange of simulation models. In *8th Int. Modelica Conf.*, Germany, Mar. 2011.

[2] M. Valasek. *Simulation Techniques for Applied Dynamics*, chapter Modeling, simulation and control of mechatronical systems. Volume 507 of Arnold and Schiehlen [8], 2008. Courses and Lectures.

[3] C. Faure, M. Ben Gaïd, N. Pernet, M. Fremovici, G. Font, and G. Corde. Methods for real-time simulation of cyber-physical systems: application to automotive domain. In *IEEE Int. Wirel. Comm. Mobi. Comput. Conf. IWCMC'11*, pages 1105–1110, 2011.

[4] A. Ben Khaled, M. Ben Gaïd, D. Simon, and G. Font. Multicore simulation of powertrains using weakly synchronized model partitioning. In *E-COSM'12 IFAC Workshop on Engine and Powertrain Control Simulation and Modeling*, Rueil-Malmaison, France, Oct. 2012.

[5] M. Sjölund, R. Braun, P. Fritzson, and P. Krus. Towards efficient distributed simulation in Modelica using transmission line modeling. In *3rd Int. Workshop on Equation-Based Object-Oriented Modeling Languages and Tools EOOLT*, pages 71–80. Linköping Univ. Electronic Press, 2010.

[6] F. Zhang, M. Yeddanapudi, and P. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *Proc. 17th IFAC World Congress*, pages 7967–7972, Seoul, South Korea, Jul. 2008.

[7] M. Arnold, C. Clauss, and T. Schierz. Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation V2.0. *Arch. Mech. Eng.*, LX(1):6–156, 2013.

[8] M. Arnold and W. Schiehlen, editors. *Simulation Techniques for Applied Dynamics*, volume 507 of *CISM International Centre for Mechanical Sciences*. SpringerWienNewYork, 2008. Courses and Lectures.

[9] M. J. Weinberger, G. Seroussi, and G. Sapiro. The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. *IEEE Trans. Image Process.*, 9(8):1309–1324, 2000.

[10] N. Wiener. *Extrapolation, interpolation, and smoothing of stationary time series*. MIT Press, 1949.

[11] R. G. Brown. *Smoothing, Forecasting and Prediction of Discrete Time Series*. Prentice-Hall, 1962.

[12] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. Probability and Statistics. Wiley, 2008.

[13] C. Faure. *Real-time simulation of physical models toward hardware-in-the-loop validation*. PhD thesis, Univ. Paris-Est, 2011.

[14] R. W. Hamming. *Numerical methods for scientists and engineers*. Dover publications, 1973.

[15] S. M. Stigler. Gauss and the invention of least squares. *Ann. Stat.*, 9(3):465–474, 1981.

[16] G. F. C. de Bruyn and J. M. de Villiers. Formulas for $1 + 2^p + 3^p + \ldots + n^p$. *Fibonacci Q.*, 32(3):271–276, 1994.

[17] M. Arnold. *Simulation Techniques for Applied Dynamics*, chapter Numerical methods for simulation in applied dynamics. Volume 507 of Arnold and Schiehlen [8], 2008. Courses and Lectures.

[18] Z. Benjelloun-Touimi, M. Ben Gaïd, J. Bohbot, A. Dutoya, H. Hadj-Amor, P. Moulin, H. Saafi, and N. Pernet. From physical modeling to real-time simulation: Feedback on the use of modelica in the engine control development toolchain. In *8th Int. Modelica Conf.*, Dresden, Germany, 2011. Linköping Univ. Electronic Press.

[19] P. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. Wiley.com, 2010.

[20] M. Ben Gaïd, G. Corde, A. Chasse, B. Léty, R. De La Rubia, and M. Ould Abdellahi. Heterogeneous model integration and virtual experimentation using xMOD : Application to hybrid powertrain design and validation. In *7th EUROSIM Congress on Modeling and Simulation*, Prague, Czech Republic, Sep. 2010.

[21] C. Chaux, J.-C. Pesquet, and L. Duval. Noise covariance properties in dual-tree wavelet decompositions. *IEEE Trans. Inform. Theory*, 53(12):4680–4700, Dec. 2007.