A Probabilistic take on system modeling with Modelica and Python

Sylvain Girard*, Thierry Yalamas

Phimeca Engineering, 18/20 boulevard de Reuilly 75012, Paris, France

By promoting modularity and automation, the system stance on numerical modeling, also deemed "0D/1D" because it focuses on dynamic macroscopic representations, is a very efficient approach to industrial problem solving, regarding both time and manpower. The coming of age of the *functional mock-up interface*, an initiative to standardize black box formulations of numerical models, encourages to bolster system modeling with a mathematical treatment of uncertainty for greater robustness.

Two practical examples illustrate how to tackle this endeavor in practice with an easy to handle set of tools centered around the Modelica and Python programming languages. First, a new Python module, Otfmi, allowing to use functional mock-up units, namely standardized black boxes, with the OpenTURNS uncertainty analysis library is introduced. Finally is expounded a method for routine inclusion in Modelica models of Python functions, for instance mathematical approximations built with OpenTURNS.

*Corresponding author: girard@phimeca.com. Discussed Python, Modelica and C code can be obtained upon request.

keywords: system modelling, 0D/1D, differential equations, Modelica, Python, probability

Introduction

Numerical models are used extensively to understand and predict the behavior of industrial systems. They occur at various stages of the industrial life cycle: whether it be to explore possible designs in conception phase, optimize operation, monitor performance and health of installations, or establish a maintenance strategy. The first part of this paper compares two archetypes of models, the component and system model, and gives an overview of current technologies pertaining to the latter.

Taking decisions, warding off risks or asserting a performance raises three kinds of challenges related to the concept of uncertainty:

- the *intrinsic variability of phenomena*, for instance when considering natural environments, chaotic systems or human factors;
- the *lack of knowledge*, in the case of "hidden" systems, or when observations are scarce or imprecise;
- the *complexity* arising from the reunion of a multitude of sub-systems, especially in networks, or when making long term projections.

Dealing with them impels to a probabilistic approach of numerical modeling which has produced a sizable corpus of combinable methods (Ghanem, Higdon and Owhadi, 2016) in the last decades. The second part of the paper explores how the more mature and widespread of these methods, sensitivity analysis (Saltelli et al., 2008), uncertainty propagation (Walker et al., 2003) and model emulation (Sacks et al., 1989), may be applied to system models.

Solving industrial problems with component and system models

Some fundamental concepts related to the motivations and ways of modeling are defined in section . It has very little epistemological pretentions: its concern is to forestall misconceptions often raised by the heavy semantic load of words such as "model" or "system". The component and system approaches to numerical modeling are then defined and compared section . Finally, section describes the Modelica programming language and associated tools for system modeling.

Purpose, constituents and structure of numerical models

A model is a representation of an observed or imagined reality, the real system. Its *purpose* is most of the time either or both explanatory, namely to make sense of the real system, or predictive, that is to supplement strictly empirical knowledge by deduction from premisses, mostly other models of assumed generic validity.

A model is a collection of *constituents* assembled by *relations*. Constituents are of three kinds:

- constants are fixed during the construction of the model,
- parameters can be altered to reflect different states of the real system or design alternatives,
- and latent constituents are actualized by an operation called *simulation*.

Mathematical models are models whose constituents and relations belong to the mathematical realm, mostly numerical variables and equations. Numerical models are mathematical models that are simulated by the numerical resolution of equations.

Simple models of an electric kettle

Let consider an electric kettle as an illustration of real system. The most elementary mathematical model is the single variable model. Consider for instance a scalar denoted T representing the temperature of the water in the kettle. This model is easily related

to observation, but lacking any relation, simulating it is somewhat trivial. Then comes the single equation equation model. A commonplace example are equations expressing conservation of mass or energy. Suppose for instance, that the industrial problem at hand is to investigate the energy consumption of boiling tap water. The following equation, directly derived from the first law of thermodynamics, describes heat transfer:

$$Q = m\,c\,\Delta T \tag{1}$$

If water is indeed the only material that the kettle will heat, then the specific heat capacity c is a constant, equal to $4181.3\,\mathrm{J\,kg^{-1}\,K^{-1}}$. The mass of water m (in kg) is a parameter. The final temperature for the boiling problem is $373.15\,\mathrm{K}$. The temperature change, ΔT, may be either a parameter if tap water temperature may vary, or a constant otherwise. There is only one latent variable, the amount of heat Q (in J), which will be simulated using equation (1). Should the industrial problem be differently formulated, say "how much can the kettle increase the temperature of tap water given an energy budget?", the status of Q and ΔT would be different. The final step of determining the nature of each constituent will be called "packing" because it results in a black box with inputs (parameters) and outputs (latent variables). In mathematical terms, a black box is nothing else than a function, which is precisely the object that most uncertainty analysis methods apply to.

Let conclude the kettle example by considering another industrial problem, the design of the heating element of the kettle. This new purpose calls for additional equations. Ohm's law gives the functioning current (I in A) as the ratio of the constant tension of mains electricity (U, most probably $120\,\mathrm{V}$ or $230\,\mathrm{V}$) by the resistance of the kettle's heating element (R in Ω), a parameter:

$$I = \frac{U}{R}. \tag{2}$$

Joule's first law equates power (P in W) to the product of tension and current:

$$P = U\,I. \tag{3}$$

Assuming a total conversion of electrical power into thermal energy, the definition of power states

$$P = \frac{Q}{\Delta t}, \tag{4}$$

where Δt is the heating duration in s. Simulating this 4 equation model shows that heating duration is proportional to the mass of water and resistance of the heating component. Here again, a different packing of the same constituents and relations would result in a model fit to solve some other industrial problems.

Causality and graphical representations

Mathematical models may be envisioned as a set of nodes each corresponding to a constituent, interlinked by set of edges, the relations. Figure 1 represents the 3 versions of the kettle model described above. Each equation is represented as a graph linking the involved constituents. Such graphical representation will be called "equation diagram" in the following. Summoning graph theory is tempting, and would probably be fruitful, but is beyond the scope of the present paper.

The 4 equation model (lower left) is an example of nested packing: delimiting black boxes inside another enclosing black box may help making sense of it. The definition of power as energy per unit of time appears as a link between a thermal and an electrical subsystems. The connection between these two black boxes is simply an equality between quantities from one sub-subsystem and the other. It may be more enlightening to adopt instead an *acausal* formulation, by making a distinction between *potential* and *flowing* quantities. Differences in potentials, for instance temperature or tension, between one side of the connection and the other is then seen to cause a transfer of flowing quantities, say heat or current, though the connection. This view points draws attention to a serious flaw in the 4 equation kettle model: the temperature of the resistor should play a role. Indeed, the water will never boil if it does not exceed 100 °C. It is equally intuitive that heat transfer should benefit from an increase in the surface of contact between the resistor and water. An acausal connection between the thermal and electrical sub-systems calls for a physical model of the interface, for instance Newton's law of cooling. This

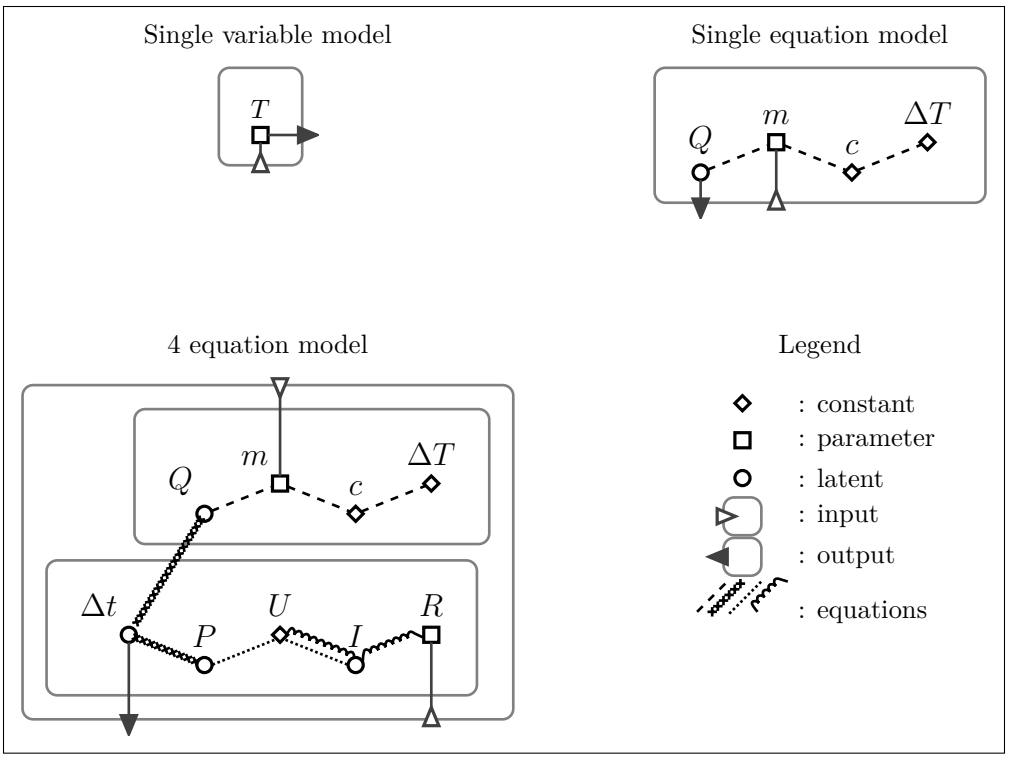

Figure 1: Equation diagrams of three models of an electric kettle of increasing complexity.

additional equation will impose constraints on the material and shape of the resistor that are actually crucial for solving the industrial problem of heating duration: the 4 equation model states that a one millimeter long copper wire, whose resistance is negligible, could boil one liter of water instantly! The concepts of potential and flows are all-pervading in physics, and so are they in the system approach to modeling described in the next section. In the following, a graphical representation of subsystems connected by links representing flows induced by potential differences will be called "flow diagram".

Component and system standpoints on the modeling process

The construction of a numerical model generally comprises three phases.

1. The *system breakdown* is the identification of sub-systems such that each
   - matches tangible or conceptual delimitations in the real system,
   - or can be described with a coherent and easy to articulate body of knowledge,
   - or is sufficient to answer some question.
2. The *mathematical formulation* is the identification of constituents and relations describing each sub-systems.
3. The *simulation setup* is twofold: after packing each sub-system as defined above comes the programming of algorithms for solving the resulting set or sets of equations.

This process may be carried out recursively, namely some sub-systems may be broken down, which results in a hierarchy of mathematical formulations. Similarly, it may appear necessary to extend the boundaries of the system. In such situation, a sub-system at the former top level of the hierarchy can be either extended, or become part of an enclosing sub-system at the new top level.

As shown by the kettle example, packing is tightly dependent on the purpose of the model. Likewise, the choice of the appropriate level for setting up simulation depends on the industrial problem at hand. In the *component* approach, packing occurs at the

lowest level, resulting in "monolithic" black boxes. Recursive packing at any higher level produces black boxes enclosing interconnected black boxes and will be called the *system* approach. The next section expands on the motivations of these two approaches.

Complex systems and difficult problems

Breaking down the kettle system into {water; heating part} may seem gratuitous sophistication. However, the relevance of the notion of subsystem becomes increasingly obvious as grows the *complexity* of the system, that is the number of its elements and their interactions.

Let substitute the kettle with a steam generator of a pressurized water nuclear power plant. A steam generator is a heat exchanger made of a vessel with a liquid water input at the bottom, and a steam output at the top. The produced steam is fed to the turbine and condenses before coming back to the steam generator: this is the secondary loop. The high pressure liquid water of the primary loop conveys heat from the nuclear reactor. It flows through the steam generator in a bundle of thousands of U-shaped small diameter tubes, entering at high temperature, say 320 °C, and exiting some 50 °C colder. The U-shaped tubes are the counterpart of the kettle's heating resistor. The nuclear fission reaction and the primary and secondary flows must be controlled to ensure that energy is steadily extracted from the reactor. Hence, instead of simply considering boiling time as in the simplistic kettle example, the dynamic of the system is now of interest. This requires extending the modeling scope to the automatic control and some of its associated actuators: pumps, valves and reactor regulation devices.

Hopefully, this detour into nuclear engineering illustrates the benefits of the system approach. First, a well thought system breakdown is a valuable conceptual support to the explanatory purpose of the model. Then, modeling a automatic control and thermal hydraulics of diphasic fluids are two very different tasks, requiring specific sets of skills and tools. From a pragmatic point of view, partitioned and hierarchical modeling promotes modularity. It enables reusing model parts in different contexts, and restricts the need to update other components when applying local modifications. Finally, an iterative system

breakdown may be the cornerstone of the modeling workflow. System extension and breakdown are two ways of supplementing a model until it is fit for its purpose. This is if one endorses the "Keep it simple stupid" (KISS) dogma, but appropriate system breakdown is equally helpful when it comes to prune a model a posteriori in a "Keep it descriptive stupid" (KIDS) approach (Edmonds and Moss, 2004).

There is another source of complication that arises when using modeling to solve real industrial problems, namely that some questions are *difficult* in the sense that they require detailed representations of physical phenomena with high predictive performance. The two main sources of difficulty are high stakes, such as safety issues, which leave small room for approximation, and certain physical phenomena that elude simple description or a fine spatial resolution, for instance the local effect of turbulence.

System complexity and problem difficulty are two forces pulling in orthogonal directions. Macroscopic questions about complex systems can be answered without inquiry of the detailed behavior of each part. As such there are ideal candidates for the system approach. Conversely, the technical (mixing diverse expertise) and practical (high computational costs) issues of interconnecting sophisticated models often implies than solely the component approach is viable for answering difficult questions.

Customary characteristics of component and system models

The component and system approaches often have differing attributes, in addition to their divergent motivations. The most prominent of them are listed in table 1 and commented below.

Answering difficult questions often necessitates simulations with a fine spatial resolution. Hence many component models represent physical phenomena in dimension 2 or 3, for instances using finite elements on a mesh. System models are usually built from a more macroscopic viewpoint and operate with punctual or averaged quantities with no or a very rough spatial discretization. The system approach is therefore called "0D/1D modeling" in some communities.

Mostly as a consequence of the dimension attribute, computation time of component

models, ranging from one minute up to many hours, are often much larger than those of system models, ranging from seconds and below to a few minutes. Hence, the technical issues of distributed and high performance computing are rarely raised by the system approach.

Solving the equations of a component model is often carried out by an ad hoc program, written for instance in FORTRAN of C++. It may also relies on a tool, for instance Salome-Meca (EDF, 2017b) or Ansys (Ansys, 2017), but still requires dedicated attention and skills, for instance for designing the mesh a in finite element computation. The numerical issues posed by system models are comparatively simpler, and can most of the time be overcome with easy to use generic purpose solvers. Actually, the main difficulty in simulating system models often lies in their *initialization*. The initial state of the model is calculated by solving an algebraic system of initial conditions with a standard algorithm such as Newton's method. It commonly requires from the modeler to provide not too far off guess values for the state variables. Even though this is at core a problem of numerical analysis, the key expertise for solving it is most often an in-depth knowledge of the real system.

As a matter of fact, the main actors of the component and system modeling approaches are differing. In addition to specialist of the real system and of the physics used to describe it, component model development usually involve programmers and numerical analysis experts for designing an ad hoc solver, or specialists of dedicated modeling tools. System modeling may be seen as a skill in itself, but in many occasions a real system can be modeled without advanced knowledge neither of programming nor numerical analysis. Hence, the system model designers are often R&D engineers well versed in the study, design or operation of the real system.

Component models, especially those with an ad hoc solver, are often complex pieces of software. As such, intervals between new versions may be long because extensive tests and verifications are required to ensure that the modifications did not break anything. Conversely, the dissociation of the physical and numerical ingredients in system modeling, as well as nested packing, allow for small localized increments and modifications. This

Table 1: Some customary characteristics of component and system models

Attribute	Component model	System model
Spatial dimension	2D, 3D	0D, 1D
Computation time	Long (minutes to hours)	Short (seconds to minutes)
Equation solver	Ad hoc method or complex tool	Generic algorithm
Actors	Physicist, programmer, experts in numerical methods, mesh generation, parallel computing	R&D Engineers specialized in the conception, operation or monitoring of the industrial systems at hand
Pace of development	Scarce but in-depth overhauls	Continuous mutation

results in a quicker rate of change and an almost continuous evolution of the model.

Modelica, a programming language for building system models

Modelica (*Modelica, A Unified Object-Oriented Language for Systems Modeling – Language Specification* 2014; Tiller, 2015a) is a language for programming numerical models which fits particularly well the system approach. It allows to represent systems by differential algebraic equations (Petzold, 1982) which are then solved by a dedicated multipurpose third-party tool.

It is mostly a declarative language, meaning that it focuses on expressing *what* the program should do, namely what equations it must solve, instead of *how*. A Modelica program contains almost exclusively equations written as they are found in textbooks, a few hints intended at the third-party solver such as initial guess values, and almost no instructions for controlling the program's flow. Organizing the equations in a way suitable for classical numerical methods is automated by the third-party tool, as well

as producing a sequential program applying those methods, and further compiling this program into executable machine code.

Modelica is object-oriented: it has objects that can be nested to represent systems and sub-systems, and objects to represent, possibly sophisticated, connections between subsystems. Hence, the conceptual structure of the model is reflected in the code. It also enables higher level abstraction layers in which the constituents are no more variables and equations but sub-systems and connectors. A set of classes representing for instance a resistor, a capacitor, an inductor and an electrical connector can be used to model electronic circuits just as on a breadboard. In such settings, properties such as Kirchhoff's circuit laws emerge implicitly from the definition of electrical current and tension in the connector class.

Designing classes to represent sub-systems is part of the packing step as defined in section . However, Modelica allows to design black-boxes without specifying their inputs and outputs, which is called *acausal* modeling. Solving algebraic loops by causality analysis and symbolic manipulation of equations is again left to the third-party tool. This means that a single program may serve several purposes just by altering the nature its constituents between simulations. Causal modeling is sometimes more appropriated, for instance to model an automatic control. Modelica supports mixing causal and acausal modeling seamlessly. More generally, it does not make any distinction of physics or engineering domains, which makes it very versatile and able to deal with hybrid systems.

Finally, Modelica is an open language. It can be used free of charges, and the open access to its specification promotes interoperability.

Third-party tools

The Modelica language specification states a vocabulary and syntax for representing differential algebraic equations. It leaves open the issue of how to actually solve them. Third-party tools, such as the commercial software Dymola (Dassault Systèmes, n.d.) or the open source OpenModelica, are used to analyze the differential algebraic equations implicitly stated in a Modelica program and produce machine code including an

appropriate solver.

While compilation and solving are the main purpose of Modelica tools, most of them are actually full fledged development environments. They notably provide a graphical interface and a graphical representation of Modelica models akin to a flow diagram. Building circuit models as in the previous example can be achieved by dragging and dropping modules and linking them with the mouse. This allows non-programmers to build numerical model using libraries of components dedicated to their domains of interest. Many such libraries are available, a selection of which are listed by the Modelica association (Modelica association, n.d.) which also maintain the vast Modelica standard library. Finally, Modelica tools often provide basic post-treatment tools, mostly for plotting time traces of simulated variables.

Applying probabilistic methods to system models

Sensitivity analysis, uncertainty propagation and model emulation all apply to mathematical functions, which unambiguously relate to numerical models considered as black boxes with inputs and outputs. Hence, applying these methods to component models is straightforward, at least conceptually, and so is applying them to system models *non-intrusively*, that is by considering only the enclosing black box. However, setting up an interface between the numerical model and the software implementing the mathematical methods is a task whose share of the global effort may become unduly substantial if it must be reiterated for each new model, or version of a model. Section presents a sets of tools automating these chores, thus allowing to concentrate one's effort on physical and statistical modeling. The efficiency of these is illustrated by a case study with a classic epidemic model.

Linking together component models is often technically difficult, or simply impractical because of computation time. Yet, using only streamlined macroscopic formulations may be frustrating when some sophisticated descriptions of high predictive power are available. Section shows on a practical example how emulation may be used to bridge

the gap between the component and system approaches.

Standard, non-intrusive approach

Applying a mathematical method to a black box numerical model involves 4 pieces of software:

- the algorithm implementing the method;
- the *pilot* algorithm, running simulations with appropriate input values;
- the *input-output data interface*, also know as the *wrapper* in the parlance of computer experiment, feeding input values to the numerical model and retrieving corresponding output values;
- the numerical model itself.

OpenTURNS is an open source C++ library of methods related to uncertainty analysis with a Python interface (Baudin et al., 2015; OpenTURNS consortium, 2017). It provides high level objects for probabilistic modeling and an extensive set of tools for data analysis, sensitivity analysis and model emulation.

Piloting simulation is generally a matter of only a few lines of code, except in the case of high performance computing which will not be dealt with in the present text.

Building the data interface is the core of the setup effort. Section shows how this can be automated with the functional mock-up interface standard and the PyFMI Python module. The freshly released Python module Otfmi integrates these tools within the OpenTURNS environment. The usability of the resulting tool set is illustrated by two practical examples.

Standardized black-boxes with the Functional mock-up interface

The functional mock-up interface (FMI) standard specifies a format for multipurpose, easy to build and reusable data interfaces (*Functional mock-up interface for model exchange and co-simulation* 2014; Modelica association, 2017a). A *functional mock-up unit* (FMU)

is a black box defined by the FMI standard, actually a zip archive containing an XML file describing the variables of the model, and a set of possibly compiled C functions required for the simulation itself. An FMU can be quasi autonomous if a solver algorithm is included among those functions. It is then fit for "co-simulation" usage, namely the interdependent simulation of several connected FMUs. It may also rely on a third-party solver, to be provided at run time. In this case, only the "model exchange" framework of the FMI standard is available.

A list of tools capable of compiling or piloting FMUs is maintained by the Modelica association (Modelica association, 2017b). The FMI lib library (Modelon AB, 2017a) is an open source low level C implementation of the FMI. Built on top of it, the Python PyFMI module (Andersson, 2016; Andersson, Åkesson and Führer, 2016; Modelon AB, 2017b,c) provides an high level interface for FMU model exchange and co-simulation.

Otfmi, a new Python module bringing FMI into OpenTURNS

The purpose of the Otfmi open source Python module (EDF, 2017a) is to promote a probabilistic approach to system models, in particular those written in Modelica, by enabling easy manipulation of FMUs within OpenTURNS. It emulates OpenTURNS interface to Python function numerical models and relies on PyFMI for its core features, namely:

1. loading an FMU as an object being integral part of OpenTURNS;
2. selecting input and output variables;
3. setting some initial values, possibly using a text file interface, to ease initialization;
4. simulating the model with samples of input values, possibly in parallel;
5. retrieving the simulation result.

Epidemic forecast with uncertainty quantification

The SIR model describes the dynamics of an epidemic by partitioning a population into 3 classes: "Susceptible" (of contracting the disease), "Infected" and "Removed" (dead or immunized). Its simplest formulation, devised by Kermack and McKendrick (1927), is a system of 3 differential equations:

$$\frac{dS}{dt} = -\frac{\beta I S}{N}, \tag{5}$$

$$\frac{dI}{dt} = \frac{\beta I S}{N} - \gamma I, \tag{6}$$

$$\frac{dR}{dt} = \gamma I, \tag{7}$$

where S, I and R respectively stand for the number of susceptible, infected and removed individuals, and N for the total population. It has two parameters, β^{-1} the typical time separating contacts between individuals, and γ^{-1}, the typical time for recovery.

Despite its apparent simplicity, this non-linear system has no analytic solution. It is however easily programmed with Modelica :

```
model SIR
  Real susceptible;
  Real infected;
  Real removed;

  parameter Real infected_initial=1;
  parameter Real total_population=763;

  input Real contact_rate;
  input Real average_infectious_period;

  initial equation

  total_population = susceptible + infected + removed;
```

```
  infected = infected_initial;

  equation

  der(susceptible) = -contact_rate * susceptible * infected /
    total_population;
  der(infected) = contact_rate * susceptible * infected / total_population -
    infected / average_infectious_period;
  der(removed) = infected / average_infectious_period;
end SIR;
```

Variables are declared before the `initial equation` keyword, the two initial equations simply set the initial population of the 3 compartments, and the 3 differential equations are directly transcribed from Kermack and McKendrick's article, after the `equation` keyword.

We used this model to forecast the evolution of an influenza outbreak in a boarding school based on the first 4 daily case counts (Anonymous, 1978). Of course, the estimates of the 2 parameters based on these scant observations spanning only the outset of the epidemic are uncertain. We built an independent sample of their posterior distribution using OpenTURNS's implementation of the Metropolis Hastings algorithm with a pseudo-likelihood derived from comparison of simulated epidemic onsets with the data. An FMU of the above program was compiled with OpenModelica, and the simulations were launched using standard OpenTURNS instructions: thanks to Otfmi, it made no difference in the Python code that the model was written in Modelica.

Figure 2 shows 50 simulations of the epidemic further evolution, illustrating the uncertainty of the forecast ensuing from the restricted available data. This uncertainty is considerable indeed, as the maximum number of case may vary by a factor of 3 from one pair of parameters to the other. This result matches the conclusion drawn by House (2014) using the stochastic version of the SIR model.

Simulating from the fourth day of the epidemic instead of the first was achieved by

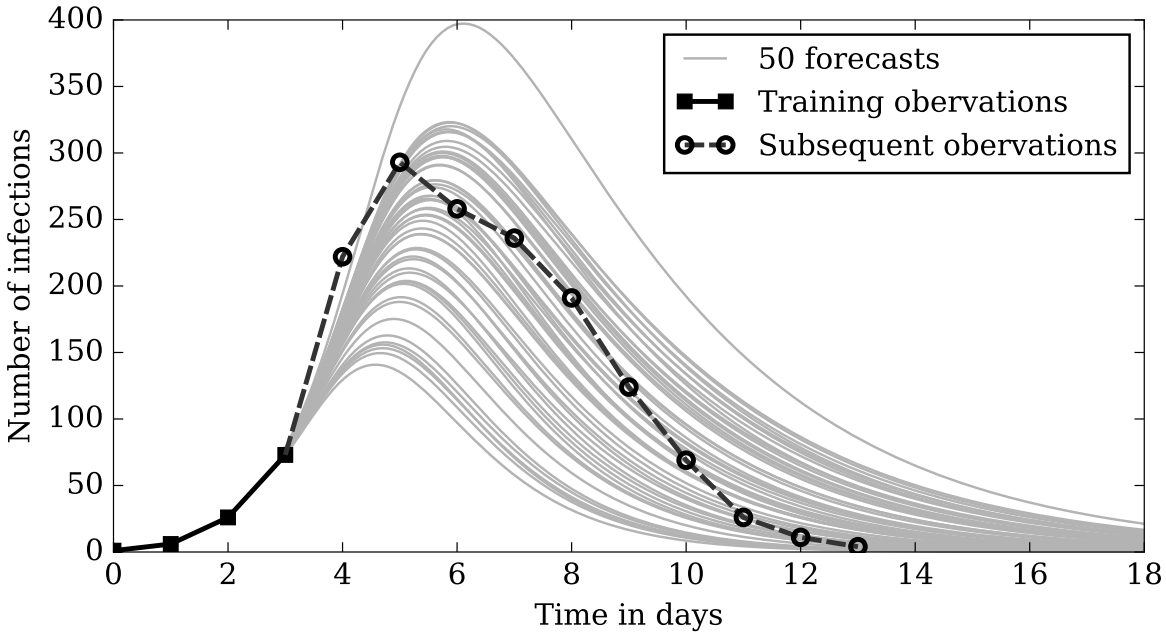

Figure 2: 50 forecasts of the epidemic evolution based on the parameter distributions inferred with the Metropolis Hastings algorithm.

slightly altering the initialization part of the Modelica model: instead of setting the initial population of the 3 compartments, we specified the initial value of the derivative of the infected population, whose value was computed from the observations.

Mixing component and system modeling: inclusion of a Python emulator in a Modelica model

Straightforward inclusion of component models into system models is rarely tractable, if only due to computation time. Model emulation is a commonplace means to circumvent computation time issues when applying statistical method. An *emulator* is a mathematical approximation of a black box numerical model. Depending on the community and used techniques, such approximations are also known as meta-models, surrogate models or response surfaces. A procedure for including Python built emulators of component models into system models written in Modelica is described below.

Let consider the real system constituted of a thermoelectric generator converting

waste heat from a vehicle combustion engine into electricity for charging a small battery meant as an ancillary source of power. A thermoelectric module is obtained by linking alternatively p-doped and n-doped semiconductors. It generates a tension when placed between cold and hot sources, : this is the Seebeck effect (Ismail and Ahmed, 2009). The objective is to estimate bounds for the average battery charging rate during 30 min drives.

Component model of the thermoelectric generator

GTeSim (Ait-Taleb, 2015; Mov'eo, 2015) is a static, 3D meshed, component model of a thermoelectric generator programmed in Matlab. It is customizable through a graphical interface and allows for batch simulations using Excel spreadsheets as input. Once the user has specified the generator architecture and material parameters, GTeSim predicts the stationary tension resulting from any given set of values of the five model inputs: the current, and the temperatures and flow rates of the hot and cold sources. Thanks to some simplifications, the simulation times are relatively short, about 4 s. However, a simulation of a dynamic system model including this component would typically necessitate hundreds of evaluations, which, in a probabilistic framework, should be further multiplied by the sample size commonly ranging from decades to thousands.

Acausal representation of the system model

The system model of the thermoelectric generator and battery is depicted in figure 3. The links connecting sub-systems represent flows induced by potential differences: electric links carry current (I_{TEG} and $I_{battery}$) between differing tensions (U_{TEG} and $U_{battery}$), themo-hydraulic links carry both mass ($\dot{m}_{hot}$ and $\dot{m}_{cold}$) between differing pressures, and heat between differing temperatures (T_{hot} and T_{cold}). The hot source is the vehicle waste heat whose temperature varies with motor usage. The cold source is the cooling system whose temperature will be assumed to stay at about 100 °C.

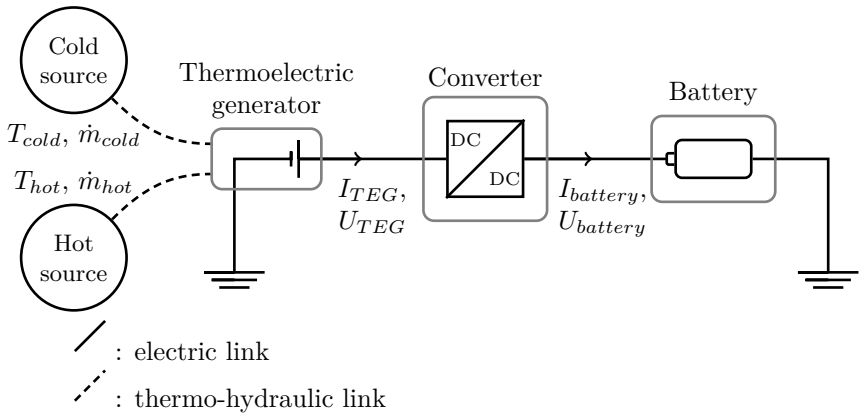

Figure 3: Flow diagram of the system model of the thermoelectric generator and battery. Links between sub-systems represent potential differences and flows.

Emulation of the component model

A gaussian process emulator (Rasmussen and Williams, 2006; Sacks et al., 1989) was built out of a training sample of 512 GTeSim simulations evenly spanning the input space. It predicts the generator current output for any given set of mass flows, temperatures and tension. Indeed, the DC-DC converter imposes a tension at its negative pin (U_{TEG}) slightly above the tension of the battery ($U_{battery}$), namely about 12 V. Hence, the tension at the positive pin of the generator is almost constant, while it is mostly the current (I_{TEG}) that responds to variations of the hot source temperature (T_{hot}). Therefore, reversing the causality of the component model produces a much simpler to solve non-linear system of equations (Tiller, 2015b).

Leave-one-out cross validation indicates that the emulator is performing very well: the resulting standardized mean squared error is 0.2 %. Admittedly, a simpler regression model would have probably been sufficient for emulating this particular model. Indeed, the current response is monotonous and very smooth, and mostly depends on only two variables without noticeable interactions, tension and temperature of the hot source, as was shown by a Sobol' (Sobol', 2001) sensitivity analysis. However, gaussian process emulation was chosen for this demonstration because of its versatility and relative ease of use: it accommodates a vast variety of contexts with limited need for user expertise.

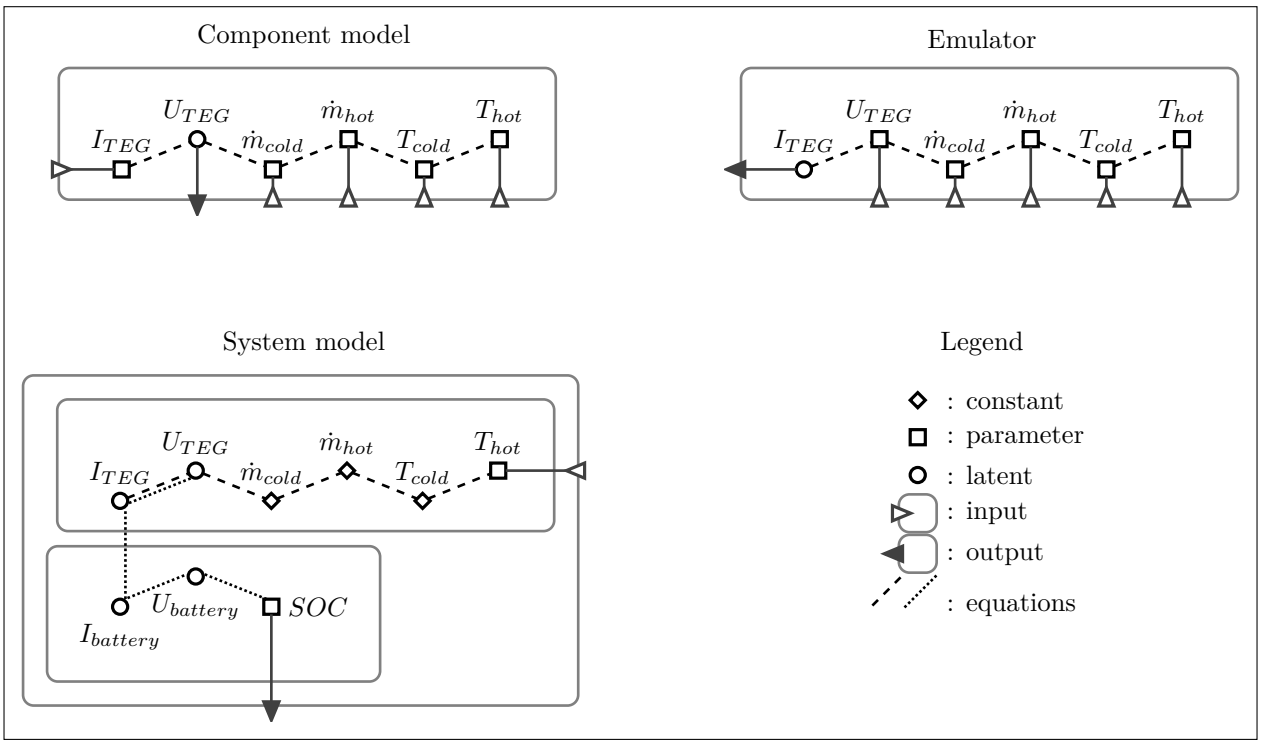

Figure 4: Equation diagram of the component model and emulator of the thermoelectric generator, and of the system model with the battery.

Causal representation of the system model

The equation diagrams of the component model and emulator of the thermoelectric generator, and of the system model including the battery can be compared in figure 4. The diagrams of the component model and emulator are almost the same but for a causality swap. The diagram of the system model represents the system of equations of the battery by a single graph. It has a single input, the temperature of the hot source, because mass flows and the temperature of the cold source are assumed constant. Currents and tensions became internal variables, and the new output is the battery state of charge (SOC).

Including foreign code in Modelica models

As a complement to acausal physical equations, procedural functions can be defined in Modelica. Such functions may be written with the Modelica language itself or rely on

external code in FORTRAN or C.

Emulators could certainly be build using either of these two languages. However, interpreted language such as Python, R or Matlab are much more accessible to non expert programmers, and are de facto the preferred tools of many applied mathematicians and engineers.

Python has an application programming interface allowing embedding of Python code within C programs (Python Software Foundation, 2017). Two programs were written for including the emulator in the system model:

- a generic purpose C program for wrapping Python functions and serve as a data interface with Modelica,
- and a very simple Modelica function which make it available for inclusion in any model.

Scripting commands for setting the appropriate compilation flags were found in the OpenModelica documentation (Open Source Modelica Consortium, 2017). The resulting code may be obtained upon request to the author.

A similar procedure could probably be applied with R using the RInside and Rcpp packages (Eddelbuettel, 2017). Otherwise, the inclusion of a gaussian process emulator built with the R package DiceKriging (Roustant, Ginsbourger and Deville, 2012) was successfully tested with the interfacing programs described above and the rpy2 Python package for embedding R (Gautier and other, 2017). Likewise, there are tools for compiling Matlab code into C (Mathworks, 2017).

While the approach presented here is fully functional and less intricate in practice than it may sound, a somewhat more wieldly procedure might be achieved by designing a FMU compiler for Python programs.

Temperature dependence of the rate of charge

The top panel of figure 5 represents a one hour signal of hot source temperature generated with prescribed low frequency variations and a noise simulated with the pulse accumulation

algorithm of Carrettoni and Cremonesi (2010). The signal has three stable stages and evolve linearly in-between: the average temperature starts at 600 °C (until 1200 s), then drops to 400 °C (1800 s to 2400 s), and climbs back to 500 °C (3000 s to the end). As stated in section , the tension is almost constant. The difference between the current output of the generator and the current feeding the battery follows the temperature variations. The currents are null when the temperature goes below 420 °C. The generator is by-passed during the lapses, indicated by gray flat tints in figure 5, when this occurs. It would otherwise function as a Peltier effect cooling unit and empty the battery.

The state of charge increases steadily during the initial and final stable stages, with a steeper slope at higher temperature. It is obviously constant when the generator is by-passed. Interestingly, the high frequency noise of the temperature signal is filtered out by the dynamics of the circuit, resulting a smooth charge of the battery.

A set of 1000 random 30 min temperature signals stationary around a 550 °C average where generated using the pulse algorithm on a wider frequency domain, 5 of which are displayed in figure 6. The corresponding battery charges were simulated after compiling the system model into an FMU. According to this probabilistic model, the average rate of charge during a 30 min drive has a mean of $8.25\,\%\,\mathrm{h}^{-1}$, symmetrically bracketed by its 5 % and 95 % quantiles at $\pm 2.20\,\%\,\mathrm{h}^{-1}$.

An additional set of 1000 signals were simulated with the same power spectral density but varying average temperatures. It revealed a linear dependence of the rate of charge with temperature: a 10 °C temperature increase induces a $0.59\,\%\,\mathrm{h}^{-1}$ increase in the rate of charge.

Conclusions and perspectives

The component and system complementary stances on numerical modeling have been discussed. They differ conceptually and in there motivations, but also on a more practical ground: they involve different people, tools and workflows. Applied mathematicians may be more customary to the component approach because it produces black boxes, which

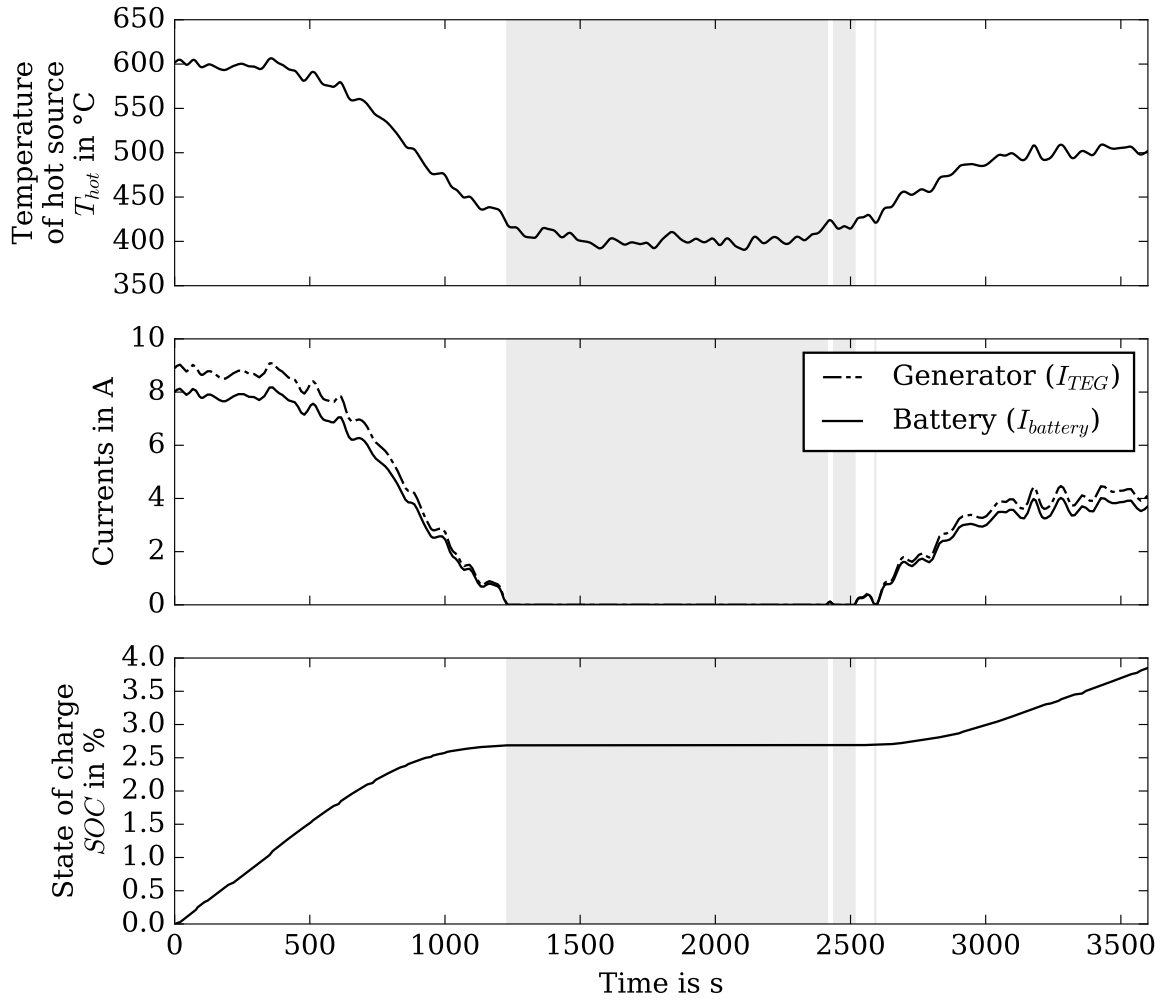

Figure 5: Input signal of the hot source temperature (top) and resulting currents (middle) and state of charge (bottom) simulated with the system model of the thermoelectric generator and battery.

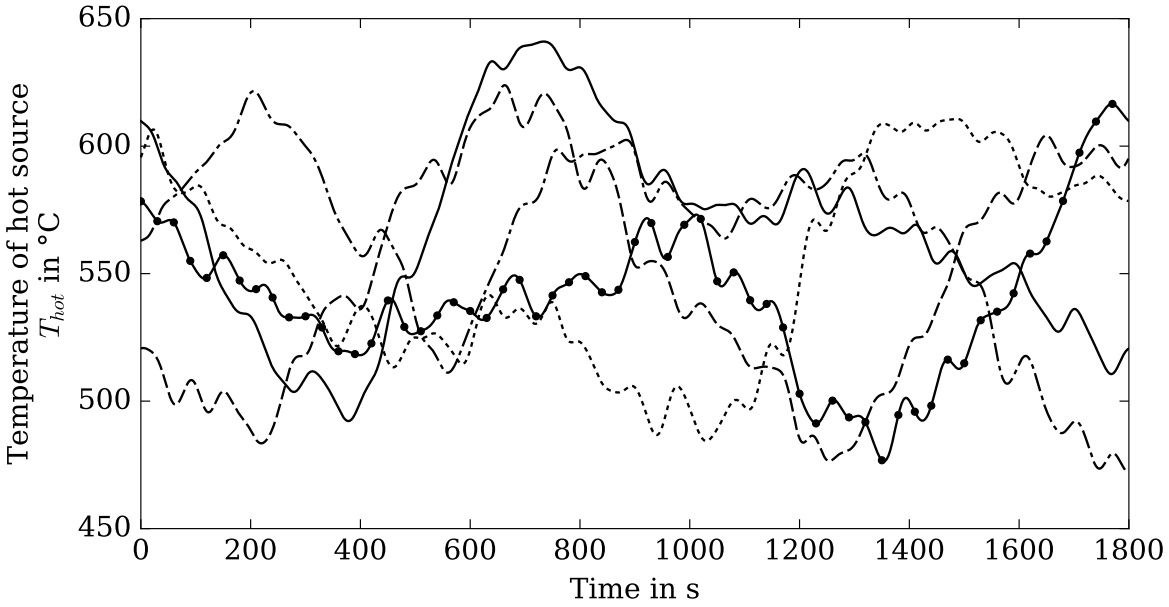

Figure 6: Five 30 min random input temperature signals.

are inherently akin to mathematical functions. Yet, the recent advent of the FMI and associated tools has considerably eased the use of probabilistic methods with system models by promoting standardization and automation. The system approach incites to cultivate modularity. Hence, the combination of the Modelica language, for designing system models, with an interpreted language with scientific computation capabilities, such as Python, enables fast paced industrial problem solving firmly rooted into mathematical rigor. To that purpose, a new Python module, Otfmi, has been developed. It provides an FMU interface within the OpenTURNS uncertainty analysis library. Meanwhile, a practical example with a thermoelectric generator model showed that probabilistic techniques could be mobilized to take advantage of the predictive power of component models within the system modeling approach.

In a similar spirit, it appears that the design of numerical models could benefit from a routine usage of probabilistic methods from the beginning of the process instead of considering them as icing on the cake, once the model has been finalized. Possible realizations of this idea include

- performing systematic sensitivity analyses of sub-systems to focus the modeling effort on the most influential ones;
- estimating states when part of the global model is still missing, or to ease initialization, using data assimilation or a Bayesian framework (Bonvini, 2017);
- emulating sub-systems for controlling complexity in a bottom up construction of large network models from local scale components.

As a matter of fact, applying probabilistic methods to sub-systems while simulating the enclosing black box soon draws attention to some current research topics. First, system models are often dynamic, which may pose some difficulties with methods that were designed with scalar inputs and outputs in mind. A common expedient to this issue is to project fields or time series on concise orthogonal bases. Principal component analysis may for instance be used to compute Sobol' indices (Girard, 2014; Lamboni, Monod and Makowski, 2010), or build an emulator (Girard et al., 2016). However, a tighter integration of probabilistic methods into the design of system models may require to take into account the dynamics more thoroughly. Some recent leads in that direction are currently investigated within the "Common Horizon of Open Research on Uncertainty in Simulations" (CHORUS) french research project (Bhattacharya, 2007; Castelletti et al., 2012; Conti et al., 2009; Liu and West, 2009).

Another issue ensues from the graph underlying a system model: the inputs of a given sub-system may be outputs of some others, and therefore be correlated. Such induced correlations are an obstacle to standard sensitivity analysis methods. In particular, Sobol' indices are ill defined when the input variables are not independent. Caniou (2012) proposed a framework for sensitivity analysis of nested models using polynomial chaos expansion and the analysis of covariance (ANCOVA), and applied it with OpenTURNS and the YACS module for managing computation workflows in SALOME (EDF, 2017c). Phimeca plans to adapt this approach to Modelica models.

Acknowledgments

The first part of this paper benefited from fruitful discussions with participants to the 2016 sessions of the "Uncertainty and Industry" working group of the Institut de Maîtrise des Risques (IMdR).

Part of the work and reflections reported here was motivated by the study of hybrid bus battery longevity undertaken within the "*Businova Evolution*" project, leaded by Safra and funded by the *French Environment & Energy Management Agency* (ADEM) as part of the *Investments for the Future* programme (PIA).

The work on the inclusion of Python functions within Modelica models was performed within the CHORUS research project.

The GTeSim generator model has been developed by Sherpa Engineering within the RENOTER joint research project (Mov'eo, 2017). Sandrine Doucet and Lahsen Ait-Thaleb provided the generator and battery system model, and kindly helped in setting it up.

References

Ait-Taleb, Lahsen (2015). *GTeSim : Simulateur Générateur Thermoélectrique – Spécification Détaillée*. Tech. rep. Sherpa Engineering.

Andersson, Christian (2016). 'Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface'. PhD thesis. Lund University.

Andersson, Christian, Johan Åkesson and Claus Führer (2016). *PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface*. Technical Report in Mathematical Sciences 2. Centre for Mathematical Sciences, Lund University.

Anonymous (1978). 'Influenza in a boarding school'. In: *British Medical Journal* 587.1. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1603269/?page=2.

Ansys (2017). *Ansys website*. URL: http://www.ansys.com.

Baudin, Michaël, Anne Dutfoy, Bertrand Iooss and Anne-Laure Popelin (2015). 'OpenTURNS: An industrial software for uncertainty quantification in simulation'. In:

Bhattacharya, Sourabh (2007). 'A simulation approach to Bayesian emulation of complex dynamic computer models'. In: *Bayesian Analysis* 2.4, pp. 783–815. DOI: 10.1214/07-ba232. URL: https://doi.org/10.1214%5C%2F07-ba232.

Bonvini, Marco (2017). *EstimationPy Python package*. URL: http://lbl-srg.github.io/EstimationPy/index.html.

Caniou, Yann (2012). 'Global sensitivity analysis for nested and multiscale modelling'. Theses. Université Blaise Pascal – Clermont-Ferrand II. URL: https://tel.archives-ouvertes.fr/tel-00864175.

Carrettoni, M. and O. Cremonesi (2010). 'Generation of noise time series with arbitrary power spectrum'. In: *Computer Physics Communications* 181.12, pp. 1982–1985.

Castelletti, A. et al. (2012). 'A general framework for Dynamic Emulation Modelling in environmental problems'. In: *Environmental Modelling & Software* 34.0, pp. 5–18. DOI: http://dx.doi.org/10.1016/j.envsoft.2012.01.002.

Conti, Stefano, John Paul Gosling, Jeremy Oakley and Anthony O'Hagan (2009). 'Gaussian process emulation of dynamic computer codes'. In: *Biometrika* 96.3, pp. 663–676.

Dassault Systèmes (n.d.). *Dymola*. URL: https://www.3ds.com/products-services/catia/products/dymola/.

Eddelbuettel, Dirk (2017). *Embed R in C++ code*. URL: http://dirk.eddelbuettel.com/code/rinside.html.

EDF (2017a). *Otfmi Python module*. This software was developed with the collaboration of Phimeca Engineering (Sylvain Girard). URL: https://github.com/openturns/otfmi.

– (2017b). *Salome-Meca presentation*. URL: http://www.code-aster.org/V2/spip.php?article146.

– (2017c). *YACS*. URL: http://www.salome-platform.org/user-section/about/yacs.

Edmonds, Bruce and Scott Moss (2004). 'From KISS to KIDS–an 'anti-simplistic' modelling approach'. In: *International Workshop on Multi-Agent Systems and Agent-Based Simulation*. Springer, pp. 130–144.

Functional mock-up interface for model exchange and co-simulation (2014). version 2.0. Modelica association.

Gautier, Laurent and other (2017). *rpy2 Python package*. URL: https://rpy2.bitbucket.io/.

Ghanem, Roger, David Higdon and Houman Owhadi, eds. (2016). *Handbook of Uncertainty Quantification*. Springer Nature. DOI: 10.1007/978-3-319-11259-6. URL: https://doi.org/10.1007%5C%2F978-3-319-11259-6.

Girard, Sylvain (2014). *Physical and Statistical Models for Steam Generator Clogging Diagnosis*. Springer International Publishing. DOI: 10.1007/978-3-319-09321-5. URL: https://doi.org/10.1007%5C%2F978-3-319-09321-5.

Girard, Sylvain, Vivien Mallet, Irène Korsakissok and Anne Mathieu (2016). 'Emulation and Sobol' sensitivity analysis of an atmospheric dispersion model applied to the Fukushima nuclear accident'. In: *Journal of Geophysical Research: Atmospheres*. DOI: 10.1002/2015jd023993. URL: http://dx.doi.org/10.1002/2015JD023993.

House, Thomas (2014). 'For principled model fitting in mathematical biology'. In: *Journal of Mathematical Biology* 70.5, pp. 1007–1013. DOI: 10.1007/s00285-014-0787-6. URL: https://doi.org/10.1007/s00285-014-0787-6.

Ismail, Basel I. and Wael H. Ahmed (2009). 'Thermoelectric power generation using waste-heat energy as an alternative green technology'. In: *Recent Patents on Electrical & Electronic Engineering (Formerly Recent Patents on Electrical Engineering)* 2.1, pp. 27–39.

Kermack, William O. and Anderson G. McKendrick (1927). 'A contribution to the mathematical theory of epidemics'. In: *Proceedings of the Royal Society of London A: mathematical, physical and engineering sciences*. Vol. 115. 772. The Royal Society, pp. 700–721.

Lamboni, Matieyendou, Hervé Monod and David Makowski (2010). 'Multivariate Sensitivity Analysis to Measure Global Contribution of Input Factors in Dynamic Models'. In: *Reliability Engineering & System Safety* 96, pp. 450–459.

Liu, Fei and Mike West (2009). 'A dynamic modelling strategy for Bayesian computer model emulation'. In: *Bayesian Analysis* 4.2, pp. 393–411. DOI: 10.1214/09-ba415. URL: https://doi.org/10.1214%5C%2F09-ba415.

Mathworks (2017). *Generate C and C++ code from MATLAB code*. URL: https://www.mathworks.com/products/matlab-coder.html.

Modelica, A Unified Object-Oriented Language for Systems Modeling – Language Specification (2014). Version 3.3 Revision 1. Modelica Association. URL: https://www.modelica.org/documents/.

Modelica association (2017a). *Functional mockup interface*. URL: https://www.fmi-standard.org/.

– (2017b). *Functional mock-up interface*. URL: https://www.fmi-standard.org/tools.

– (n.d.). *Modelica libraries*. URL: https://www.modelica.org/ModelicaLibrariesOverview.

Modelon AB (2017a). *FMI library*. URL: http://www.jmodelica.org/FMILibrary (visited on 10/02/2017).

– (2017b). *PyFMI on PyPI*. URL: https://pypi.python.org/pypi/PyFMI (visited on 14/02/2017).

– (2017c). *PyFMI Python module*. URL: http://www.jmodelica.org/page/4924 (visited on 10/02/2017).

Mov'eo (2015). *Produits issus des projets R&D*. Tech. rep. Mov'eo. URL: http://pole-moveo.org/wp-content/uploads/2015/07/MOVEO%5C_BOOK%5C_Produits-Projets-RD%5C_Success-Stories%5C_2015.pdf.

– (2017). *RENOTER research project*. URL: http://pole-moveo.org/en/projets/renoter/.

Open Source Modelica Consortium (2017). *Calling external Python Code from a Modelica model*. URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/

interop%5C_c%5C_python.html%5C#calling-external-python-code-from-a-modelica-model.

OpenTURNS consortium (2017). *OpenTURNS*. URL: http://openturns.org/ (visited on 14/02/2017).

Petzold, Linda R. (1982). *Description of DASSL: a differential/algebraic system solver*. Tech. rep. SAND-82-8637; CONF-820810-21. Sandia National Laboratories, Livermore, CA (USA).

Python Software Foundation (2017). *Embedding Python in Another Application*. URL: https://docs.python.org/release/2.7.9/extending/embedding.html.

Rasmussen, Carl Edward and Christopher K. I. Williams (2006). *Gaussian process for machine learning*. MIT press.

Roustant, Olivier, David Ginsbourger and Yves Deville (2012). 'DiceKriging, DiceOptim: two R packages for the analysis of computer experiments by kriging-based metamodeling and optimization'. In: *Journal of Statistical Software* 51.1, pp. 1–55.

Sacks, Jerome, William J. Welch, Toby J. Mitchell and Henry P. Wynn (1989). 'Design and Analysis of Computer Experiments'. In: *Statistical Science* 4.4, pp. 409–423. DOI: 10.1214/ss/1177012413. URL: http://dx.doi.org/10.1214/ss/1177012413.

Saltelli, Andrea et al. (2008). *Global sensitivity analysis: the primer*. Wiley Online Library.

Sobol', Il'ya Meerovich (2001). 'Global Sensitivity Indices for Nonlinear Mathematical Models and their Monte Carlo Estimates'. In: *Mathematics and Computers in Simulation* 55, pp. 271–280.

Tiller, Michael M. (2015a). *Modelica by Example*. Xogeny. URL: http://book.xogeny.com/.

– (2015b). *Modelica by Example: about non-linear systems*. Xogeny. URL: http://book.xogeny.com/behavior/functions/nonlinear/.

Walker, Warren E et al. (2003). 'Defining uncertainty: a conceptual basis for uncertainty management in model-based decision support'. In: *Integrated assessment* 4.1, pp. 5–17.