POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A Benchmarking Framework for Performance Evaluation of Modelica Compilers

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Author: **Marina Nikolic**

Student ID: 913154
Advisor: Prof. Giovanni Agosta
Co-advisors: Francesco Casella, Daniele Cattaneo, Stefano Cherubin,
Alberto Leva, Federico Terraneo
Academic Year: 2020/2021

Abstract

Industry utilises simulation software to reduce costs, during system design (reducing the need for physical prototypes) and also during operation, e.g. for diagnostic purposes. There are various tools on the market, each with their respective language: some are specific to a single domain (e.g., electronics) while others are domain-neutral (physical simulations of different kinds). Among the latter, the most used language is Modelica. The description of a model, often through equations, must be translated into simulation code. The compiler has the task to make the transition from a language to the other possible, and the impact of the time spent and memory necessary makes the simulation (and compilation) of large models unfeasible with the current Modelica tools. This thesis has the purpose of enriching HiPerMod, a framework for benchmarking Modelica compilers, with a test case that needs a solver able to handle differential and algebraic equations, some of which nonlinear. The test cases include hand-written C++ code that can be taken as an ideal model for the performances of existing simulators. This new benchmark has been embedded in the existing project, and the simulation and compilation times of the C++ version have been compared with the performances of the leader-of-the-market open-source Modelica software, OpenModelica. Spacial and temporal complexities have been evaluated, and the compilation time complexity has been discussed as well. It has been shown how the hand-written code, keeping the array structure of the data of the equations and the for loops, allows to reduce the simulation time and the memory size of the executable file. Moreover, we show how OpenModelica's compilation time depends on the number of scalar equations of the model, while keeping the data structures and for loops allows for constant compilation time.

Keywords: Benchmarking, Modelica, Compilers, Simulation Software, Modelling

Abstract in lingua italiana

Il mondo dell'industria utilizza i software di simulazione sia durante la progettazione (riducendo il bisogno di prototipi fisici), sia per scopi diagnostici durante il funzionamento. Sono presenti sul mercato tool di vario genere, ognuno con il rispettivo linguaggio: da quelli specifici a un singolo ambito (ad esempio, l'elettronica) a quelli più neutrali (simulazioni fisiche di vario genere). Tra questi ultimi, il linguaggio più utilizzato è Modelica. La descrizione di un modello, spesso attraverso equazioni, deve essere tradotta in codice di simulazione. Il compilatore ha il compito di rendere possibile il passaggio da un linguaggio all'altro, e l'impatto del tempo speso e della memoria necessaria rendono impossibile la simulazione (e compilazione) di grandi modelli scritti in Modelica con gli strumenti attuali. Questa tesi ha come obiettivo arricchire HiPerMod, un framework per il benchmarking dei compilatori del linguaggio Modelica, con un caso di test che richieda un risolutore di sistemi di equazioni differenziali e algebriche anche non lineari. È stato scritto il codice che funge da modello ideale per le prestazioni dei simulatori esistenti, incorporato nel progetto esistente, e confrontato con le prestazioni del software open-source leader del mercato, OpenModelica. Sono valutate le complessità spaziale e temporale, ed è discussa la complessità del tempo di compilazione del codice sorgente. Si mostra come il codice scritto a mano, mantenendo la struttura ad array delle equazioni e i cicli for, permette di ridurre il tempo di esecuzione e la memoria occupata dal file eseguibile. Inoltre, si mostra come il tempo di compilazione di OpenModelica dipende dal numero di equazioni scalari del modello, mentre mantenendo la struttura dei dati e i cicli for questo rimane costante.

Parole chiave: Benchmark, Modelica, Compilatori, Software di simulazione, Modeling

Contents

Introduction

Simulation of a system Simulation of physical systems has brought numerous advantages into many fields of academic and industrial research. The possibility of simulating a system has reduced development time and costs, and it allows to produce prototypes more likely to fulfil the requirements. Hence the need for domain-specific languages to describe such systems, and for tools to simulate them - faster and cheaper, but simple enough to use for experts of any field.

Modelica One of the languages allowing to describe physical systems is Modelica. It is used in many fields, and, for this reason, its target users are not software developers but the same engineers that are tasked with the development of such system.

The Modelica standard library, which is a collection of commonly used physical models, covers many fields: from electronics to mechanics, but also control systems and hierarchical state machines. Moreover, it is always possible to model any system described with equations. It is particularly suited for system-level simulation (i.e. simulation of large cyber-physical systems, sometimes not fully specified). Improving the performances of Modelica compilers and simulation engines would impact the other fields of research that make use of the language, particularly for large-scale systems.

Applications Modelica is nowadays used by many automotive companies to design energy-efficient vehicles, including Ford, General Motors, Toyota, BMW.

Power plant providers also make use of this language. Recently, Modelica has been considered for the simulation of large-scale power generation and transmission systems, and new libraries have been created for that purpose.

The most used tools are Dymola, which is proprietary, and OpenModelica, free and open-source. Dymola is used for simulations related to automotive, aerospace, defence, energy, and modelling of industrial equipment among other things. OpenModelica is used for power systems, plant optimisation, and water treatment among other things.

Modelica tools Modelica is a declarative language that models physical systems or phenomena through equations. It is implemented in tools often made by whole environments for simulation and analysis. The core of these tools, however, is the compiler. The main purpose of such compilers is translating a declarative language into imperative code. The equations describing a model need to be manipulated before being simulated and often solved through numerical integration. Most of the tools make use of an external solver which has been developed independently. It's the case of the SUNDIALS suite, containing, among others, the IDA solver. This thesis focuses on a benchmark test case using IDA to solve a non-linear differential algebraic equation system.

Benchmark suites The presence of many simulation environments, each with its compiler, stimulates the development of benchmark suites to compare their performances and results. Differently from other fields, the time-to-solution from a Modelica model to the results' file takes into account the compilation time. Since models are supposed to be simulated only once in many usage scenarios, the compilation time becomes as relevant as the simulation time. Changing a parameter within the model means re-compiling the whole model from scratch. Hence, simulation and compilation are always executed together - and only once for each model, for the results are to be stored in a file and later retrieved for analysis purposes.

Existing tools often show worse performances in the first analysis of the model and compilation than the simulation itself, particularly in the case of very large models. The simulation code is produced after flattening the models, i.e. getting rid of the complex data structures or arrays, as well as of any hierarchical structure. It often brings an increase in the number of equations (e.g. an array equation is transformed into a series of identical scalar equations). It brings more complexity to the analysis of the model. Each equation is often mapped into a function of sequential code, and each of those functions has its own header, environment and variables. Identical functions are not merged into one to be called with different arguments, causing a worse space complexity of the simulation code and executable file.

Document structure In this thesis, we enrich a benchmark framework for Modelica compilers with a test case using an external DAE solver, IDA. The first chapter contains the state of the art and some theoretical background, that is the Modelica language specification and the tools used for the benchmark tests. In the second chapter, the model used in the benchmark is presented. The PowerGrid model is part of the HiPerMod benchmark suite, and the purpose of this thesis is to create an equivalent simulation in C++ code. The implementation details of the code are given in chapter three. The evaluation and

the comparisons with a leader-of-the-market existing tool - that is, OpenModelica - is presented in the fourth chapter. We also show how the experiments are automated. Finally, the last chapter of this thesis will summarise the work and propose further developments in the field of Modelica compilers.

1 | State of the Art

1.1. Mathematical background

A system of equations can be often used to describe a physical phenomenon. An equation can be differential or algebraic, depending on whether there are differentiation operators or not.

System of equations ODE stands for Ordinary Differential Equation. The term *ordinary* is in contrast with the term *partial* derivative that may be with respect to one or more variables. ODE refers to a system of equations written in the form

$$\dot{x} = f(x, t) \tag{1.1}$$

where the derivatives of the functions x are explicit with regard to the function f depending on x and t only.

DAE stands for Differential-Algebraic Equation. It is a generalisation of an ODE, and it has form

$$F(x, v, \dot{x}, t) = 0 \tag{1.2}$$

where $\dot{x}$ is the vector of derivatives of x, and v is a vector of algebraic variables whose derivatives do not appear in F.

A semi-explicit DAE is shown in equation 1.3.

$$\begin{cases} \dot{x} = f(x, v, t) \\ g(x, v, t) = 0 \end{cases} \tag{1.3}$$

An IVP, initial value problem, is made of an ODE or DAE and an initial condition which specifies the value of the unknown function and its derivative at a given point. Equation 1.4 shows the general form of an IVP. F is the (vectorial) residual function of the DAE (or ODE), y an $\dot{y}$ are the vectors of variables and derivatives. t_0 is the point in time to

which the known values refer to.

$$\begin{cases} F(t, y, \dot{y}) = 0 \\ y(t_0) = y_0 \\ \dot{y}(t_0) = \dot{y}_0 \end{cases} \tag{1.4}$$

Residual function It is always possible to express any system in the form 1.2. It is trivial, in fact, to transform an equation expressed as in 1.1 and obtain a new F as:

$$F(x, \dot{x}, t) = f(x, t) - \dot{x} \tag{1.5}$$

The function F is called *residual function.*

Jacobian matrix Some numerical methods need a Jacobian matrix to solve a system of equations. The *Jacobian matrix* of a system of equations is the matrix of the partial derivatives of each equation with respect to each variable. The rows represent the equations and the columns the variables, so that an element $J_{i,j}$, the element in the i-th row and j-th column is the partial derivative of the i-th component of the residual function with respect to the j-th variable.

Equation 1.6 shows the Jacobian matrix of a system with n equations and m variables. If the number of variables and equations is the same, then the Jacobian matrix is square (it has the same number of rows and columns).

$$\mathbf{J} = \begin{bmatrix} \frac{\partial F_1}{\partial v_1} & \cdots & \frac{\partial F_1}{\partial v_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial v_1} & \cdots & \frac{\partial F_n}{\partial v_m} \end{bmatrix} \tag{1.6}$$

Index of a DAE The *index* of a DAE measures the distance from a DAE to its related ODE. There are many index definitions, one of which is the differentiation index. The *differentiation index* of a DAE system is the number of times needed to differentiate the DAE to get an ODE.

Explicit and implicit methods There are several methods to solve DAEs, mostly depending on the properties of the system. The analysis of discrete systems or discrete-time systems is out of the scope of this thesis. Here we discuss only continuous-time systems with continuous input and output signals.

Explicit methods compute the state of a system at a later time from the state of the system at the current time. An example of an explicit method is the forward Euler method, which substitutes the derivative of the functions with the finite differences formula [5].

Implicit methods solve an equation involving both the current state of the system and the next one, instead of computing the next state directly from the current. Among the implicit solution methods for systems of equations, there are:

- **Backward Euler** It is the most basic numerical method to solve ODEs. It computes the approximations using

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}) \tag{1.7}$$

  which differs from the explicit forward Euler method in that the latter uses $f(t_k, y_k)$ [11].

- **Backward Differentiation Formula** BDF is a family of methods for the integration of ODEs. They are linear multi-step methods that, for a given function and time, approximate the derivative of that function using information from already computed time points, thereby increasing the accuracy of the approximation [13].

- **Runge-Kutta** Runge-Kutta methods are a family of implicit and explicit iterative methods used in temporal discretisation for the approximate solutions of ODEs.It has been designed by Carl Runge and Wilhelm Kutta around 1900.

1.2. Equation-based languages for simulations

Academic and industrial research often requires studying the behaviour of a physical system before building a prototype. The mathematical analysis permits to reduce the cost and time needed for the construction of prototypes. With the advent of computers and the increase of computational power, the simulation of a virtual model became possible. The use of simulation tools increased the precision and opportunities of the initial part of the study of a system. A natural way to describe systems, e.g. in control theory, is through differential and algebraic equations, but it is not they only one.

A large portion of the simulation software currently on the market is domain-specific. Let us look at three representative examples: CSSL, ACSL, and SPICE.

The Continuous System Simulation Language (CSSL) is suitable for the simulation of dynamic systems described by ODEs [6]. ACSL stands for advanced continuous (system) simulation language [26], and it was designed for modelling and evaluation of systems de-

scribed by continuous, time-dependent, non-linear differential equations. Block diagrams are developed on-screen from pre-defined PowerBlocks representing ACSL statements, operators, functions, and/or user-defined blocks in an unlimited hierarchical structure. ACSL code is generated directly from the graphical model.

SPICE stands for Simulation Program with Integrated Circuit Emphasis [28]. It is a language designed for the simulation of electrical circuits. To describe a circuit, the user gives a name to the element with the initial letter depending on the type of element (e.g., if the user is defining a capacitor, the name must begin with C). Then, the user sets the nodes the element is connected to and the value of its specific property. For example, for a capacitor, such property is the capacity. The circuit can be simulated, and the results plotted to be analysed.

Domain-specific tools achieve high performances in their domain, but they can't model elements of another sector in a reasonable or performant way. To solve this issue, the Modelica language has been designed [25], unifying the concepts of domain-specific and earlier domain-neutral or multi-domain systems such as ASCEND[30], Dymola[14], VHDL-AMS[21].

1.3. The Modelica language

Modelica is an equation-based, object-oriented, multi-domain language for simulation of physical systems [24]. It is suitable for a large range of applications, both industrial and academic. It was designed for the study of complex systems made of elements from different domains.

A model in Modelica is defined as a set of parameters, variables, and equations [27]. Initial values are defined with initial equations, which might need to be solved before proceeding further with the compilation and simulation of the model.

Listing 1 shows a simple DAE system with two variables and one parameter. As it can be seen, the model contains first a set of parameters and variables, possibly associated with their initial value. The set of initial equations describes the system at time zero (or, more generally, at the initial time). The initial system might be different from the system described with the equations. Finally, the equation section describes the system evolution with differential and algebraic equations. In particular, the model in listing 1 has one algebraic variable and one state variable.

The object-oriented nature of the Modelica language permits the designer to define an element and re-use it throughout the model.

Listing 1 A simple DAE system

```
model FirstModel
    parameter Real a = 0.5;
    Real v(start=0.5);
    Real t;
    initial equation
        t = a;
    equation
        t = v*a;
        der(v) = t*v;
end FirstModel;
```

Listing 2 Arrays in Modelica

```
    model A //model A in file A.mo
        Real input_v;
        Real X;
    equation
        der(X) = input_v;
    end A;

    model B //model B in file B.mo
      parameter Real  par = 5.2 "some  parameter";
      A[2]  vars "an  array of  models A";
    equation
      vars[1].input_v = par;
      vars[2].input_v = vars [1].X;
    end B;
```

It is possible to define vectors of variables and models. The notation for vectors is shown in listing 2. Access to an element of the vector happens through square parenthesis. Also shown in listing 2, one model can embed other models.

Libraries containing frequently-used models are made available to the modeller. For example, when modelling circuits, all the elements (resistances, capacitors, transformers, etc...) are defined in the Modelica Standard Library. They can be instantiated in a new model and connected with the command `connect`. Listing 3 shows the use of `connect` and of the elements in the library `Modelica.Electrical.Analog`. It describes a simple RC circuit with a resistor and a capacitor connected to a generator.

It is possible to use `for` loops to define vector equations and exploit the arrays. Listing 4 is an example of the use of loops. The array variable T is the temperature of a wire

Listing 3 Using Modelica libraries' models and the construct *connect*

```
model RC "A resistor-capacitor circuit model"
  parameter Modelica.Units.SI.Voltage vb=24 "Battery voltage
     ";
  parameter Modelica.Units.SI.Resistance rr = 100;
  parameter Modelica.Units.SI.Capacitance cc = 1e-3;
  Modelica.Electrical.Analog.Basic.Ground G;
  Modelica.Electrical.Analog.Basic.Resistor r1(R=rr);
  Modelica.Electrical.Analog.Sources.StepVoltage V(V=
     vb,startTime=1);
  Modelica.Electrical.Analog.Basic.Capacitor c1(C=cc);
equation
  connect(r1.n,c1.p);
  connect(r1.p,V.p);
  connect(V.n,G.p);
  connect(c1.n,G.p);
end RC;
```

heated at one end with a heat source with constant temperature T_{hi}, and the other end is exposed to the room temperature T_{lo}. The for-loop allows declaring the equations of all the variables in T (except for the edges of the wire). These equations are all similar one to another. Also, this way, changing the value nx (and hence the granularity of the simulation) doesn't require any changes to the code. Modelica tools typically don't allow choosing a parameter at run-time, so the value nx is a constant when the model is compiled. However, simulations might need to be carried out with different combinations of parameters. In this example, changing the granularity of the model wouldn't require adding or removing the corresponding equations. However, the new model would need to be re-compiled (more about this in section 1.4).

Annotations allow setting simulation parameters in the model directly. For example, the annotation in listing 4 sets the start and stop times, the tolerance and interval values for simulation. They are instructions for the simulation and do not include data about the model's behaviour. They are used extensively to provide all kinds of information about models [27].

Listing 4 A Modelica example with for loops. Heat conductivity in a wire made of metal, heated on one side with a heat source of given temperature T_{hi}

```
model Thermal1D
    parameter Integer nx = 100;

    parameter Real g = 0.00314785;
    parameter Real c = 0.2707936;
    parameter Real Thi = 400 + 273.15;   // higher T
    parameter Real Tlo = 20 + 273.15;    // lower T

    Real T[nx](each start = Tlo);

equation
    c*der(T[1])  = g*(T[2] - T[1]) + 2*g*(Thi - T[1]);
    c*der(T[nx]) = g*(T[nx-1] - T[nx]) + 2*g*(Tlo - T[nx]);

    for x in 2 : nx-1 loop
        c*der(T[x]) = g*(T[x-1] - T[x]) + g*(T[x+1] - T[x]);
    end for;

    annotation(experiment(StartTime = 0, StopTime = 100000,
       Tolerance = 1e-6, Interval = 20));
end Thermal1D;
```

1.4. Tools for Modelica simulation

Tools for Modelica simulation are often whole environments which support modelling, compilation and simulation, data analysis. Modelica software can be divided into two groups: production-grade tools and partial works. Among the former, another distinction is between open-source and proprietary software. In this section, we present the most relevant examples of software for Modelica simulation.

1.4.1. Compilation and simulation

The processing of a Modelica file has several steps.

- **Transformation** of the equations (which are descriptive and do not impose an

ordering among themselves) into a sequence of instructions that will produce the simulation (i.e. the code of a program, typically written in C++ or C).

- **Resolution** of the DAE system. Depending on the type of system, there are several methods for numerical integration. There are also solvers on the market that implement some of those algorithms. Examples of solvers are DASSL and IDA, both solving non-linear systems. The model is firstly evaluated and then transformed into an intermediary representation that can be processed further.
- **Simulation** of the model and retrieval, analysis and plotting of the results.

The main challenge for the compiler is the transition from declarative to imperative behaviour, which does not happen with other languages. Given that there is no input to the execution of the sequential code, the simulation is often executed only once, and the results are stored for future use.

It is common, in Computer Science, to disregard the compilation time analysis because it is considered a sunk cost, irrelevant to the user of the software. However, the Modelica tools on the market produce code specific to the model file, and changing a single parameter requires the whole procedure to be repeated, from the analysis and resolution to the compiling and simulation. In this context, the compilation time impacts the total time-to-solution as much as the simulation time does.

Not only compilation time impacts every run, but it also depends on the size of the input, as it has been shown in the literature when analysing the performance of Modelica tools [2]. For these reasons, we analyse both compilation and simulation time, as well as the size of the compiled executable.

Most software environments for Modelica do not implement all the code for the solution of the system internally but use external packages (for example, Sundials - see section 1.6). The solvers are hereby integrated into the environment.

Modelica compile-time complexity is usually linear (or more than linear) w.r.t. the number of variables and equations of the model. The cause of this is the *flattening* process which unrolls loops, transforming vector equations into scalar equations and causing the loss of any hierarchical structure of the components in a model [1]. It is a known problem that flattening causes an increase of the compilation complexity, but the algorithms used for structural analysis and symbolic manipulation of the system equations can treat scalar equations only.

New algorithms have been designed to avoid or delay the flattening phase as much as possible. It has been analysed, for example, how to modify the Pantelides index reduction

algorithm to make it re-use the repeating structures of the code. The original algorithm is not able to use and preserve structures, but the new version allows to perform the Pantelides index reduction for large models independently of how long is the chain of repeated structures [4].

1.4.2. Open Source software

OpenModelica OpenModelica[1] is the most well-known and impactful work related to Modelica simulation among the open-source software. It is maintained by the OSMC (Open Source Modelica Consortium) and aims at providing long-term development and support for the modelling and simulation environment. The research topics the OSMC is carrying out are language design, symbolic and numerical algorithms and others. One research activity that is related to compilers is multi-core parallel code generation [32]. Even though OpenModelica might not be the most performing tool on the market, it represents the most relevant open-source effort in the Modelica community so far. OpenModelica covers most parts of the Modelica language, differently from most of the other tools.

1.4.3. Proprietary software

JModelica JModelica[2] is a partially open-source platform for compilation and simulation of Modelica models. It is made of several packages:

- Assimulo is a Python package for simulation.
- PyFMI is a package for loading and interacting with Functional Mock-Up Units for model exchange and co-simulation.
- FMI library is a software package written in C that enables the import of FMUs in applications. FMUs are compiled models compliant with the Functional Mock-up Interface [3].
- Modelica compiler and optimisation tools. This part of the toolkit has been moved from open to closed source code (contrarily to the formerly mentioned packages which are still open-source and available on GitHub).

Dymola The most relevant proprietary tool is Dymola[3] (DYnamic MOdeling LAboratory) [15]. It uses the Modelica libraries (both the open-source and some commercial

[1] openmodelica.org
[2] JModelica.org
[3] www.3ds.com/products-services/catia/products/dymola/

libraries) and supports the FMI standard[4]. It is based on the Dynamic Modeling Language (also called Dymola), which can be viewed as the first definition and implementation of the Modelica language [14].

1.4.4. Other works

Modia Modia[5] is an alternative to using Modelica. It is an extension of the Julia[6] [9] language for modelling and simulation of physical systems [16, 17]. Julia is an object-oriented programming language designed for high performance. It is compiled via LLVM [22], generating efficient native code, but it uses just-in-time compilation, which is not always good for performance. In fact, the time required to load a third-party package can become significant. For example, in Modia, the time to load the simulation packages ranges between several seconds to nearly a minute.

Modia represents an implementation of Modelica in Julia, its purpose being a playground for testing new features and fast prototyping. The models are described by equations, just like in Modelica, and it implements an object-oriented approach. Modia allows defining array equations, which aren't flattened during symbolic transformation. This generates more compact and efficient code. The definition of an array cannot be split among statements. Some algorithms to transform a DAE system into an ODE system are used in Modia [29]. The simulation of the model returns the solution in a Julia data structure (a matrix or array).

The Julia language has an API that permits the connection of the Julia scripts to OpenModelica (OMJulia) [23]. OMJulia is simply a wrapper over OpenModelica, so there is nothing new regarding its performance.

MARCO Previous work from our research group [1] introduces the idea of writing a new Modelica compiler from scratch with performance as the main goal. The new compiler would use the LLVM [22] compiler framework and transform the Modelica code into an intermediate representation that retains structural information from the original code, allowing the generation of machine code that is optimised for the target hardware architecture (instead of generating intermediate C code as OpenModelica does) and allowing faster compilation. The main goal of the research is to preserve arrays, loops and, in general, the object-oriented structure of the original model. To make this possible, the idea is to use a single object for the function to compute the residuals of DAE, which has

[4] fmi-standard.org/
[5] www.github.com/ModiaSim/Modia.jl
[6] www.julialang.org

to be generated only once and then called using different inputs and outputs each time it is needed.

A prototype of an optimised compiler has been designed [7, 8] and tested with a thermal conduction model compiled in OpenModelica and Dymola against hand-written C++ code. The results show that the compilation time of the Modelica code grows with the number of equations, while the hand-written code has nearly constant compile time. As for the time to solution, it has been shown that the speedup of the C++ code grows with the size of the model. Finally, the space complexity of the model is shown to be constant for the C++ code and linear for Modelica. The high performance of the prototype code is due to the absence of the flattening phase, which results in a smaller model (as it maintains its hierarchical structure). Another issue discussed is that each equation is mapped into a C function, resulting in larger code size and overhead due to function calls. Together with the loop unrolling used in the existing compilers, this causes the presence of a host of functions that are essentially the same function with different arguments.

Recently, new algorithms which preserve for-loops and the array structures of the variables have been proposed, in particular for the matching phase. It has also been proved that the problem of matching and scheduling of the equations is NP-hard if the loops are preserved [18]. Also, the compiler infrastructure has been improved through the use of MLIR, and the implementation of functions has been added to allow imperative code inside models[31].

1.5. Benchmarks for Modelica compilers

Benchmarks suites for Modelica compilers are subject of active research.

In [19], a benchmark for large models is described. The performances are tested using Python scripting to see how different tools scale with the model size. The compilers tested are OpenModelica, JModelica and Dymola. The simulation results show that, usually, Dymola performs better than the open-source tools, but the time complexity is almost the same (second or third power of the number of the equation, depending on the model and the compiler).

Most of the existing benchmark suites can't help with time complexity analysis, for it needs the possibility to change the size of the input (the number of equations). The ScalableTestSuite benchmark library was designed for this purpose [12]. The ScalableTestSuite benchmark doesn't offer an ideal target for Modelica tools, but it allows to analyse time complexity of different models.

HiPerMod [2] is a benchmark suite for Modelica compilers that aims at enabling the comparison between Modelica automatically-generated simulation code, and hand-written C++ simulation code. The C++ code is optimised by hand, thus can be considered an ideal target for compiler performance. The test cases will allow studying the time (and space) complexity by setting a parameter for the simulations (defining the model size). Each benchmark will be independent from the others, and the optimised code will be helpful for the design of high-performance compilers in the future. The full suite will cover different model examples and different solvers, in order to be used as a reference for future development of Modelica tools. The results of a simple model - solved with explicit methods - show that the C++ code tends to be one order of magnitude faster. Before this work, the benchmark models available with both Modelica file and C++ code were all solvable with explicit methods only.

1.6. Sundials package

SUNDIALS[7] stands for SUite of Nonlinear and DIfferential/ALgebraic equation Solvers. It consists of six solvers, each for a different type of problem. The suite is released open-source under a BSD license. Its goal is to provide robust time integrators and non-linear solvers to be incorporated into existing simulation tools. The solvers are designed to require minimal information from the user and to allow the user to supply their data structures, as well as to allow for user-supplied linear solvers to be integrated easily.

CVODE and CVODES are used for ODE systems. The latter includes sensitivity analysis capabilities (hence, the S). CVODE solves the initial problem for stiff and non-stiff ODEs given the explicit form. It includes, among others, BDFs methods and makes use of multi-step methods.

ARKODE is another solver for initial problems; it makes use of different methods with respect to CVODE (Runge-Kutta).

IDA and IDAS solve DAE systems. The latter includes sensitivity analysis. IDA and IDAS require a residual function provided by the user. They use BDF methods and are written in C but derived from a package written in Fortran.

KINSOL solves non-linear algebraic systems. It is written in C, but interfaces for Fortran are provided as well.

All the mentioned solvers allow the user to provide their data structures, together with the operations on those structures. The code comes with default vector structures for

[7] computing.llnl.gov/projects/sundials

both serial and distributed memory parallel environments. All parallelism is contained within the vector structures and their operations, as the same code is executed in both cases.

1.6.1. IDA solver

IDA is a package written in C. It solves the Initial Value Problem (IVP) with a DAE and the initial conditions for variables and derivatives known [20].

Before integrating a DAE system, IDA requires that the vectors with variables and derivatives are both initialised to satisfy the DAE residual equation:

$$F(t, y, \dot{y}) = 0 \tag{1.8}$$

If the user can't provide such values, there are algorithms to compute consistent initial conditions for semi-explicit, index-one problems [10]. The information needed for such algorithms to work are included in the Jacobian and residual functions already required to solve the time-dependent DAE systems.

The function `IDACalcIC(void* ida_mem, int icopt, realtype tout1)` computes the initial conditions of the problem. The argument `tout1` refers to the first time for which a solution for the DAE is requested. The integer `icopt` can have two values:

- `IDA_YA_YDP_INIT` directs IDA to compute the components of y and the differential component of $\dot{y}$ given the differential components of y. The differential components (state variables) are signalled using the function `IDASetId(void* ida mem, N_Vector id)`, where the vector `id` contains `0.0` in place of algebraic variables and `1.0` in place of differential variables.
- `IDA_Y_INIT` directs IDA to compute the components of y given the values of $\dot{y}$.

The programmer needs to implement the function for computing the residuals. Some modes also require the user to provide a Jacobian function. There is an option for IDA to compute the Jacobian numerically, but numerical robustness can be impaired. The time complexity increases greatly, and it is feasible only with smaller systems.

When initialising the data structures, the user provides to IDA two vectors for variables and derivatives respectively. The structure of the problem is lost because IDA does not require any description of the variables. The structure of the variables (scalars, vectors, or complex numbers) can still be used in the residual or Jacobian functions.

The programmer has to initialise the linear solver and link it to IDA. A non-linear solver

Listing 5 C header of user-provided residual function

```
int residualFunction(   real_t tt,
                        N_Vector yy, N_Vector yp,
                        N_Vector res,
                        void *data)
```

Listing 6 C header of user-provided Jacobian function

```
int jacobianFunction(   realtype tt, realtype cj,
                        N_Vector yy, N_Vector yp,
                        N_Vector rr,
                        SUNMatrix Jac,
                        void *user_data, N_Vector tmp1,
                        N_Vector tmp2, N_Vector tmp3
                    );
```

might be allocated by the user as well. Both the solvers might be provided by the user and not among the default alternatives, as long as they provide the methods needed for the solution of the system.

Listing 5 shows the header of the residual function IDA expects to receive. The variable `real_t tt` is the time of the simulation. The three N_Vector variables contain variables, derivatives, results. `void* data` is a pointer to the user-defined data structure IDA provides. It can store whatever data the user needs to keep track of, and it is allocated during the initialisation of IDA.

The return value of the function shall be 0 if the residual function ends the computation without errors. The residuals are written in the `N_Vector res`. Every component of the vector will store the residual of a single equation, and the corresponding line in the Jacobian matrix will store the partial derivatives of that equation w.r.t. each variable.

The Jacobian function needs to provide code for the computation of

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}} \tag{1.9}$$

Listing 6 shows the header of the user-provided Jacobian function.

The variable `realtype tt` is the time of the simulation. `realtype cj` refers to the α parameter for the computation of the Jacobian function in formula 1.9. The `N_Vectors` provided contain the variables, the derivatives, the current results of the residual function, in this order. `SUNMatrix Jac` is where the resulting Jacobian matrix is saved. `void*`

`user_data` is the pointer to the user-defined data structure IDA provides. The last three vectors are temporary variables provided for the user, e.g. in case there was a need to store partial results.

The return value shall be 0 if there are no errors.

Regarding the `SUNMatrix Jac`, IDA accepts three types of data structures to store the Jacobian matrix: dense, sparse, and band. Each option works with specific linear and non-linear solvers.

Dense Jacobian matrices are represented through two-dimensional arrays accessible like any other. They are preferable for dense systems, i.e. systems where the Jacobian matrix non-zero elements are more than the zero elements.

Sparse matrices are preferred when the variables appearing in every equation are few w.r.t. the number of equations. In this case, the growth of the number of non-zero elements is linear with respect to the number of variables. To use sparse matrices, the programmer or compiler needs to estimate an upper bound of the non-zero elements of the Jacobian matrix.

Finally, banded matrices are used with systems where the equations can be divided into subsets whose Jacobian sub-matrix is dense, but where the non-zero elements w.r.t. variables out from the dense sub-set are rare.

2 | The PowerGrid model

As we said, the HiPerMod benchmark suite has been designed to fill the absence of a benchmark suite that can compare the performance of existing and future tools to a hand-written, optimised C++ code. This suite, in its current state, contains several test cases, but not yet a scalable (in the number of equations), hand-written C++ code which uses a DAE solver for the simulation. One of the benchmark tests, the PowerGrid model, contains a DAE system with non-linear equations, thus requiring a non-linear solver. The C++ code is written without flattening the vector variables and exploiting the for-loops as much as possible.

In this chapter, we present the PowerGrid model. It is a DAE system with vectorial equations, some of them non-linear w.r.t. the variables. It is solvable using the IDA solver from the SUNDIALS package.

The PowerGrid model (see Appendix A) describes an electrical circuit with the elements (generators, and power-consuming resistors) arranged as in a matrix (thus, the name grid). Figure 2.1 represents the currents of the power grid. Circles are generators while squares are the power-consuming loads. Nodes exchange current vertically and horizontally through transmission lines. Each node has a voltage and a current output. The vertical and horizontal line currents depend on the nodes' voltages. Only generators have power generated and requested, angle and velocity.

Listing 7 shows the nested for loop which defines the currents and the loop with the current balance equations of the nodes in the first row of the grid.

Listing 8 shows the state equations of the model. The *omega* variables represent the

Listing 7 Horizontal-line current equation.

```
for i in 2:N - 1 loop
  i_n[i, 1] + i_h[i, 1] + i_v[i, 1] - i_h[i - 1, 1]
                                    = Complex(0) "Top";
end for;
```

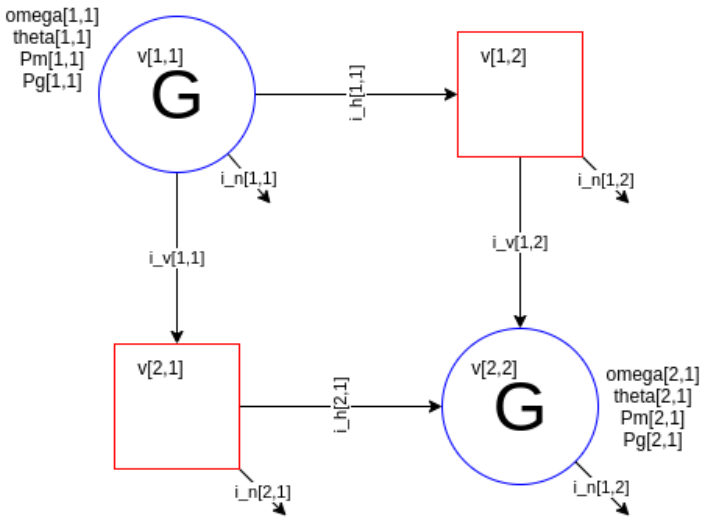

Figure 2.1: The power grid with 2 generators and 2 power-consuming units.

Listing 8 Differential equations of the PowerGrid model.

```
for i in 1:N loop
  for j in 1:Ng loop
    der(theta[i,j]) = (omega[i,j] - 1)*omega_n;
    Ta * omega[i,j] * der(omega[i,j]) =
                Pm[i,j] -
                (omega[i,j] - 1)/sigma - Pg[i,j];
  end for;
end for;
```

angular velocities of the generators, the *theta* variables represent the angles.

Currents and voltages are complex numbers, so `PowerGridDAE.mo` includes the description of the complex type with the definitions of overloaded operators on complex numbers (including conjugate, multiplication, sum, etc... used in the model).

The parameter describing the model size is N_e, the number of generators per row (or the number of even rows). There is the same number of power consumers and generators in every row and in every column. A generator's neighbours are only loads, and a resistor's neighbours are only generators.

N is the number of rows and columns in the grid. There are, in total, N^2 elements (or nodes). Every element has a voltage and a current. The elements exchange current with their neighbours on the same row and those on the same column. The total number of variables hereby introduced is $2N^2+2(N-1)N$. All these variables are complex numbers, hence using two real variables to store each. There is a set of variables for the generators

only: the phase, the derivative of the phase (angular speed), the active power output, and the power input request. There are $4N_eN = 2N^2$ real variables. Hence, the total number of variables (and equations) is:

$$N_v = 2(2N^2 + 2(N-1)N) + 2N^2 = 10N^2 - 4N \approx 10N^2 \tag{2.1}$$

Given that the parameter we decided to set is N_e, the relationship between the total number of variables in 2.1 and the number of generators in each row is:

$$N_v(N_e) = 10(2N_e)^2 - 4(2N_e) = 40N_e^2 - 8N_e \approx 40N_e^2 \tag{2.2}$$

Table 2.1: Number of equations for the tests

N_e	N	N nodes	N. equations
1	2	4	32
2	4	16	152
4	8	64	$\sim$ 600
8	16	256	$\sim$ 2500
11	22	484	$\sim$ 4750
16	32	1024	$\sim$ 10K
23	46	2116	$\sim$ 21K
32	64	4096	$\sim$ 41K
45	90	8100	$\sim$ 81K
64	128	16384	$\sim$ 163K
90	180	32400	$\sim$ 324K
128	256	65536	$\sim$ 655K

A model with one generator has 32 equations. A model with 158 generators per row has almost a million variables.

Setting the parameter N_e is done so that the total number of equations doubles from one to the next. In table 2.1, we show the values of N_e and the respective number of nodes and variables (equations) for which we ran the tests.

3 | Implementation

In this chapter, we discuss the implementation details of the C++ code simulating the PowerGrid model. First, we present the initialisation of the IDA solver and then the notable details of the residual and Jacobian functions provided. The model uses a sparse matrix of SUNMatrix type to store the Jacobian values, so the section 3.3.1 is dedicated to the description of the data structure. Finally, we show the integration of the test case with the rest of the benchmark suite.

To simulate the PowerGrid model with IDA, two functions need to be provided: one computing the residuals, and one for the Jacobian. The residual function will store the residuals in an NVector of the same size as the variables. Each element of the vector will be associated with an equation, and this association should be taken into account when computing the Jacobian. The Jacobian function will compute and store the values in a data structure, SUNMatrix, which can be dense, sparse or banded. For this model, the sparse version is used. Both functions return a control integer value which should be 0 if no errors occur. All the pointers to input and output data structures and the variables are passed as arguments.

The residual function is trivial to compute. We chose to associate the state variables to the differential equations which contain their derivatives. As for the algebraic equations, we chose to associate them with variables of equal size to better exploit the possibility to use for loops. For example, the current balance equations' residuals are stored in the respective position of the current variable of the node they refer to.

The performance analysis takes into account three measures: time complexity of the simulation, time complexity of the compilation, size of the file containing the code for the simulation.

The compilation time is relevant for this experiment because it is always part of the overall computation time from the Modelica code to the simulation results. In fact, it is not possible to use the same code generated from a model with different parameters to simulate a similar model. Every change in the model will require a new execution of the Modelica compiler. The size of the file containing the simulation code is important

Listing 9 Initialisation of IDA data structures and environment

```
1  void PowerGrid_ida::initIDA() {
2    mem = IDACreate();
3    IDASetUserData(mem,(void*)this);
4    IDAInit(mem, residualFunction, T0, variables_v,
       derivatives_v);
5
6    IDASStolerances(mem, rtol, abstol);
7
8    sunindextype nnz = 5*numberOfVariables; //upper bound
9
10   A = SUNSparseMatrix(numberOfVariables,numberOfVariables,
       nnz,CSR_MAT);
11   LS = SUNLinSol_KLU(variables_v,A);
12   IDASetLinearSolver(mem,LS,A);
13
14   IDASetJacFn(mem,jacobianFunction);
15
16   NLS = SUNNonlinSol_Newton(variables_v);
17   IDASetNonlinearSolver(mem, NLS);
18 }
```

because it impacts not only the compilation time but also the memory usage. One of the known limits of OpenModelica is the memory consumption which makes the simulation of large models practically unfeasible.

3.1. Initialisation

Listing 9 shows the method initialising all the components of IDA needed for the PowerGrid model simulation. After allocating an instance of IDA and its memory block `mem`, we associate its user-defined data structure to our PowerGrid object. A pointer to this user-defined data structure (the instance of PowerGrid, in this case) will be made available to the user in the methods computing Jacobian matrix and residuals.

After providing the residual function (described in section 3.2), the initial time, the vectors of variables and derivatives, in line 6, we set the tolerances. `rtol` is the relative tolerance, while `abstol` is the absolute tolerance. Both have value 10^{-6}.

Lines 8-10 allocate a SUNMatrix to store the values of the Jacobian matrix. Such a matrix is sparse, with one row and one column for each variable (or equation). `nnz` is the number of non-zero elements expected. In line 8, we compute an upper bound for

`nnz`. Five is the maximum number of elements - variables or their derivatives - in a single row (variables appearing in an equation), hence the maximum number of possible non-zero partial derivatives in an equation. It is necessary to provide an upper bound of the number of non-zero elements because it impacts the space allocated for the matrix. In case the number is overestimated, there will be some unused space. In the opposite case, with less space than needed, the simulation won't work.

The solvers are allocated in lines 11 and 16. The solver used is `SUNLinearSolver_KLU`. The non-linear solver is `SUNNonlinearSolver_Newton`.

Finally, in line 14, a Jacobian function is provided. The function is described in section 3.3.

3.1.1. Initial condition

The Modelica code of the model provides an initial value for every variable, algebraic or differential. IDA requires an initial condition satisfying, for both variables and derivatives, the following equation:

$$F(t_0, y, \dot{y}) = 0 \tag{3.1}$$

Listing 10 is used to compute the algebraic components of y and the differential components of $\dot{y}$. It needs a vector `N_Vector id` with value 0 associated to algebraic variables and 1 associated to state variables.

The `UserData_t` datatype used in line 3 is a custom datatype used to access the variables (and derivatives) without manually computing their index every time. The definition of the data type can be seen in listing 11. It is a struct containing pointers to the first element of each vector variable. The `ArrayView2D` class was constructed to access the multi-dimensional (two-dimensional) arrays of variables.

To use the `UserData_t` datatype, the function `UserData_t PowerGrid_ida::getUserData (N_Vector variables)` is called. The function assigns to every pointer of the data structure the reference to the first element of the corresponding vector. The position of the first element of each vector depends on the number of variables (i.e. on the size of the grid).

In the specific case of the PowerGrid model, the generators all rotate at a speed corresponding to 50 Hz frequency, and all the machine angles are initialized to zero. This would be an equilibrium condition for an infinite grid, that expanded indefinitely in all directions, since each load is equally surrounded by four generators, and the system would

Listing 10 Initial condition computation with IDA

```
int PowerGrid_ida::ICcalc(){
    N_Vector id = N_VNew_Serial(numberOfVariables);
    UserData_t id_p = getUserData(id);
    for (int i = 0; i < numberOfRows; i++){
        for (int j = 0; j < numberOfGenerators; j++){
            id_p.theta(i,j) = 1.0;
            id_p.omega(i,j) = 1.0;
            id_p.Pg(i,j) = 0.0;
            id_p.Pm(i,j) = 0.0;
        }
        for (int j = 0; j < numberOfRows; j++){
            id_p.i_n(i,j) = complex_t(0.0,0.0);
            id_p.v(i,j) = complex_t(0.0,0.0);
        }
        for (int j = 0; j < numberOfRows-1; j++){
            id_p.i_h(j,i) = complex_t(0.0,0.0);
            id_p.i_v(i,j) = complex_t(0.0,0.0);
        }
    }
    IDASetId(mem,id);
    return IDACalcIC(mem, IDA_YA_YDP_INIT, tout);
}
```

Listing 11 UserData_t datatype definition

```
typedef struct Data {
  Data(){}
  ArrayView2D<real_t> theta = ArrayView2D<real_t>(nullptr,
     0, 0);
  ArrayView2D<real_t> omega = ArrayView2D<real_t>(nullptr,
     0, 0);
  ArrayView2D<real_t> Pg = ArrayView2D<real_t>(nullptr, 0,
     0);
  ArrayView2D<real_t> Pm = ArrayView2D<real_t>(nullptr, 0,
     0);

  ArrayView2D<complex_t> i_n = ArrayView2D<complex_t>(
     nullptr, 0, 0);
  ArrayView2D<complex_t> i_h = ArrayView2D<complex_t>(
     nullptr, 0, 0);
  ArrayView2D<complex_t> i_v = ArrayView2D<complex_t>(
     nullptr, 0, 0);
  ArrayView2D<complex_t> v = ArrayView2D<complex_t>(nullptr,
      0, 0);

} UserData_t;
```

be fully symmetric. Since there are border effects due to the finite extent of the grid, the equilibrium values of the machine angles are not all equal. Hence, a transient ensues, during which angles and frequency show damped oscillations, until the system reaches equilibrium condition. This transient doesn't correspond to a real operation scenario, because it never happens in real life that all machine angles are equal. However, it is very simple to set up in the C code and, most importantly, it has about the same qualitative behaviour no matter what the size of the system is, hence it is particularly appropriate to compare the simulation performance for different system sizes.

3.2. The residual function

Listing 12 shows the header of the residual function. The static method `residualFunction` is the one used in the initialisation of IDA in listing 9, line 4. The parameters of the model have been declared as private, as well as the variables and derivatives - even though the latter are present in the arguments of the method. To use the private variables of an object, the function can't be static. At the same time, it is not possible to use a reference to the object PowerGrid's method. To by-pass this problem, the static function has the parameter `void* data` (not present in the `residual` function). Such pointer is a reference to the instance of class `PowerGrid` containing all parameters, variables and methods for the simulation.

Listing 12 Header of the residual function

```
int residual(real_t tt, N_Vector yy, N_Vector yp,
                N_Vector res, void *data);

static int residualFunction(real_t tt, N_Vector yy, N_Vector
    yp, N_Vector res, void *data) {
    PowerGrid_ida *thisObject = (PowerGrid_ida *)data;
    return thisObject->residual(tt, yy, yp, res, data);
}
```

Listing 13 shows part of the computation of the residual functions. A for loop is used to compute the residuals of the current balances of the external rows and columns (except for the corners). For the central part of the grid, a nested for loop is used, exploiting the structure of the data. Also, current balance equations involve complex numbers: in a single line of code, the residuals of two equations are defined.

Listing 13 Part of the residual function computing the residual of the current balance for the first row of the grid (except for the nodes in the corners).

```
for (int i = 1; i < numberOfRows-1; i++)
    residuals.i_n(i,0) = variables.i_n(i,0)
                        +variables.i_h(i,0)+variables.i_v(i,0)
                        -variables.i_h(i-1,0);
```

Listing 14 Header of the Jacobian function

```
int jacobian(real_t tt, realtype cj,
             N_Vector yy, N_Vector yp,
             N_Vector res, SUNMatrix JJ, void *data,
         N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);

static int jacobianFunction(real_t tt, realtype cj,
                        N_Vector yy, N_Vector yp, N_Vector res,
                        SUNMatrix JJ, void *data,
                        N_Vector tempv1, N_Vector tempv2,
                           N_Vector tempv3) {
    PowerGrid_ida *thisObject = (PowerGrid_ida *)data;
    return thisObject->jacobian(tt, cj, yy, yp, res, JJ,
       data, tempv1, tempv2, tempv3);
  }
```

The correspondent computation of partial derivatives, in the Jacobian function, is shown in listing 15, in the following section.

3.3. The Jacobian function

Listing 14 shows the header of the Jacobian function. The static method `jacobianFunction` is set as the function computing the Jacobian matrix in listing 9, line 14. For the same reasons as for the residual function, the pointer to the static function is passed to IDA, and the static method calls a non-static function to compute the values. The matrix which will contain the partial derivatives is of type `SUNMatrix`. It has been initialised as a sparse matrix in CSR format (see listing 9, line 10). The structure and means of access to a sparse matrix are explained in the following section.

3.3.1. Sparse matrix to store the Jacobian

Figure 3.1[1] shows a graphic representation of how values are stored in a sparse matrix of the SUNMATRIX_PARSE type.

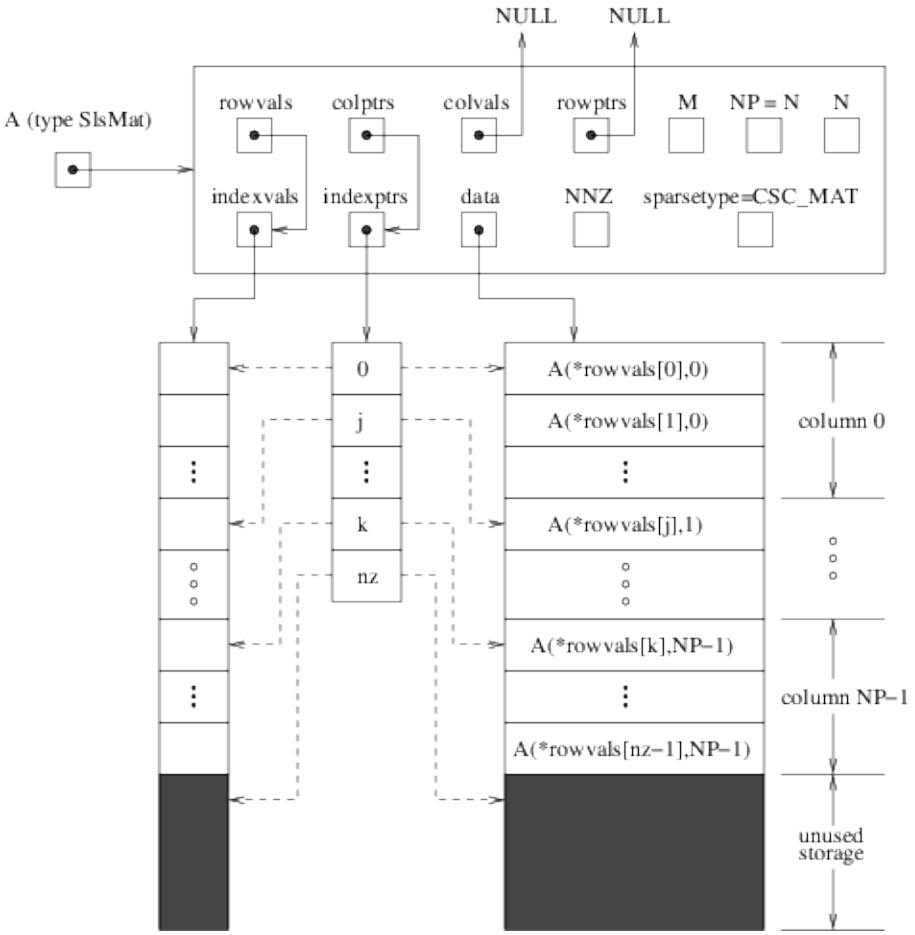

Figure 3.1: Storage of the compressed-sparse-column matrix of SUNDIALS sparse type.

There are two types of sparse matrices in the SUNMATRIX package: CSR and CSC, according to whether the non-zero values of the Jacobian matrix are stored per row or column. It will be shown in the next chapter that OpenModelica uses a CSC (compressed sparse column) type. For a matter of convenience and an easier-to-read code, we use the CSR (compressed sparse row) matrix type.

The matrix uses three arrays to store the data which will be used to reconstruct the Jacobian matrix. One of the arrays will store, for each row, how many elements are present in the matrix. More precisely, the array pointed by `indexptrs` contains the indexes of the first element referring to the current row. There is an index for each row.

[1] http://runge.math.smu.edu/arkode_dev/doc/guide/build/html/sunmatrix/SUNMatrix_Sparse.html

E.g., the number in the second cell `j` refers to the index of the first element of the second row (row with index 1). The second array, pointed by `data`, contains the values of the partial derivatives. The array pointed by `indexvals` stores the column indexes of the correspondent data values.

The insertion of new data must be performed row by row, keeping the order of the rows defined in the residual function.

Listing 15 shows the computation of the Jacobian regarding the current balance equation of the nodes in the first row of the grid. The first part is for the real part, and the second group of lines is for the imaginary part. Listing 16 shows the definition of `element` and `endr` functions. The first inserts the column index in the `indexvals` array (in this case referring to columns, in the CSC representation, one inserts the row index) and the value in the `data` array. The latter command updates the `indexptrs` array with the number of elements written till that point.

The `complex<T>` type in C++ stores the real and imaginary parts one after the other. An array of complex numbers doesn't change the order of imaginary and real parts, so the Jacobian has to be computed in that same order: for every equation, firstly the real and then the imaginary part, because IDA expects an array of real double-precision numbers and can't see the `complex_t` data structure.

3.4. Integration with the HiPerMod benchmark suite

The integration of the PowerGrid test case into the HiPerMod benchmark suite happens through the `modelSolutionI` interface. `PowerGrid` extends the interface. There are four methods to be provided: constructor, destructor, `bool step()`, and `void emitDataTo( Emitter<CSV_emitter> *outputStream)`.

The constructor and destructor methods contain respectively the initialisation and the destruction of the objects (freeing the memory used by the solver).

Function `step` will be called by HiPerMod to execute the simulation. Listing 17 shows the function implemented in the PowerGrid class. The return value shall be `true` unless the simulation has crossed the end time.

The function `void emitDataTo(Emitter<CSV_emitter> *outputStream)` is used to print the results on a file. Using the methods provided by the `Emitter` class, we can print the selected variables. In this case, the *omega* of the corner generators and the v of the corner nodes.

Listing 15 Part of the Jacobian function, computing the derivatives of the current balances of the nodes in the first row of the grid.

```
    // residuals.i_n(i,j) = variables.i_n(i,j) +
       variables.i_h(i,j) + variables.i_v(i,j) -
       variables.i_h(i-1,j) - variables.i_v(i,j-1);
    for (int j = 1; j < numberOfRows - 1; j++) {
      element(indexes.i_n(i,j).real() , 1.0);
      element(indexes.i_h(i-1,j).real() , -1.0);
      element(indexes.i_h(i,j).real() , 1.0);
      element(indexes.i_v(i,j-1).real() , -1.0);
      element(indexes.i_v(i,j).real() , 1.0);

      endr();

      element(indexes.i_n(i,j).imag() , 1.0);
      element(indexes.i_h(i-1,j).imag() , -1.0);
      element(indexes.i_h(i,j).imag() , 1.0);
      element(indexes.i_v(i,j-1).imag() , -1.0);
      element(indexes.i_v(i,j).imag() , 1.0);

      endr();
    }
```

Listing 16 Definition of *element* and *endr* calls.

```
  *rowptrs++ = 0;

  auto element=[&](int col, real_t val) {*colvals++=col; *
     jacobianV++=val;};
  auto endr=[&] (){*rowptrs++ = colvals - colvalsbegin;};
```

Listing 17 Step function.

```
bool PowerGrid_ida::step(){
  steps++;
  int r = IDASolve(mem,
            tout, &tret,
            variables_v, derivatives_v,
            IDA_ONE_STEP);
  //exception r != IDA_SUCCESS
  assert(r==0);
  return (tret < tout); //false if tret>tout
}
```

4 | Evaluation

The C++ hand-written code for the simulation of the PowerGrid model needs to be correct and scalable. Before analysing the time and space complexities, we prove that the benchmark has the first two proprieties. Then, we proceed by showing the results of our analysis. We consider both compilation and simulation time, and we also discuss the space complexity (of the compiler, hence the size of the output executable files). We briefly discuss the space complexity of the user-provided functions produced by OpenModelica.

4.1. Correctness of the results

To ensure the correctness of the obtained results, the comparison between the results of the C++ hand-written code and of the OpenModelica tool doesn't need to be done with each and every equation.

First of all, the grid modelled is symmetric with respect to the first diagonal. In fact, the variables of the element in position (n, m) will have the same values as those of the element (m, n) if we exchange i_h with i_v. Transposing the grid will have no effect.

Secondly, most of the equations in the model are algebraic. There are only N^2 state variables (and differential equations). The rest are all algebraic. The state variables are the angular positions (relative to each other) and the angular speeds of the generators. The angular speed is the derivative of the angular position.

Both in OpenModelica and in the C++ code, we chose to write in the output file the angular speed (*omega*) of the generators in the four corners: the first and second-last node of the first row, and the second and last node of the last row. We also write the voltage (real and imaginary part) of the nodes in the corners (be it a generator or a load node).

Figure 4.1 shows how the angular speeds *omega* are symmetric with respect to the main diagonal of the grid. The data are from the OpenModelica output of the model with 4 generators per row (64 nodes, around 600 equations). The same happens with the results of the C++ code.

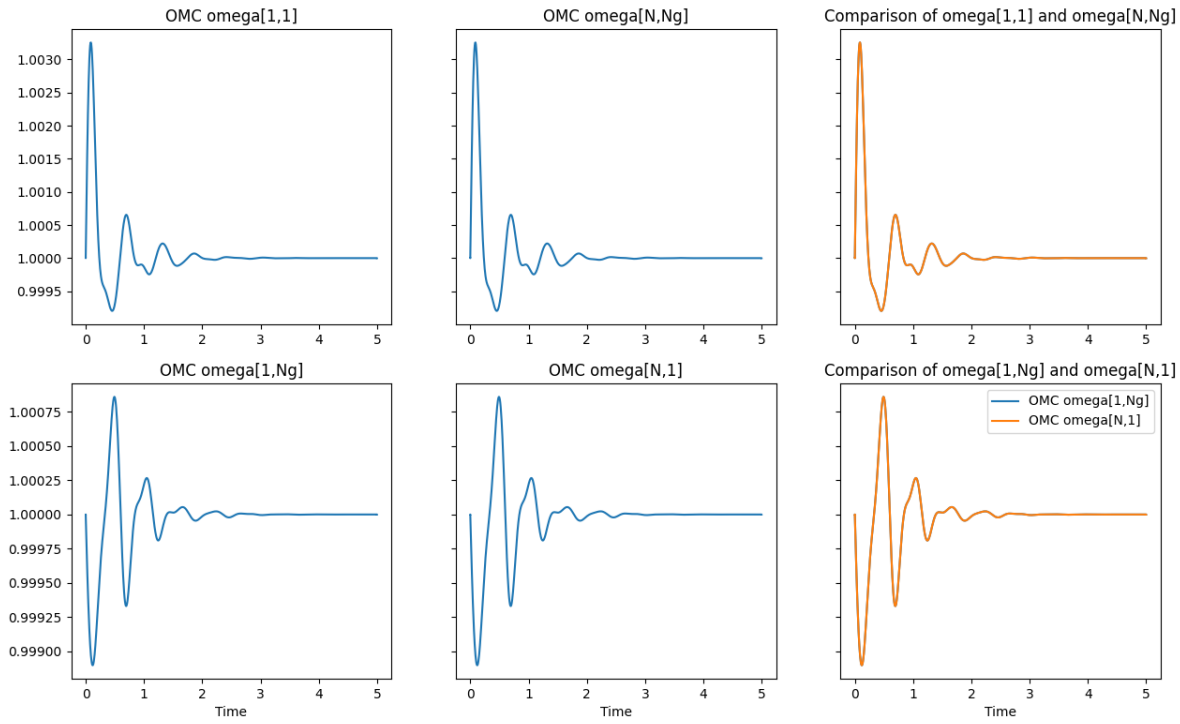

Figure 4.1: The angular speed of the four angle generators, symmetric w.r.t. the main diagonal of the grid.

To prove the correctness of the C++ hand-written code, we analyse the results for the same model with 64 total nodes. Figures 4.2 and 4.3 show the comparison of the two outputs each on a plot and one next to the other (since the two are almost identical, one of the outputs covers the other on the plot).

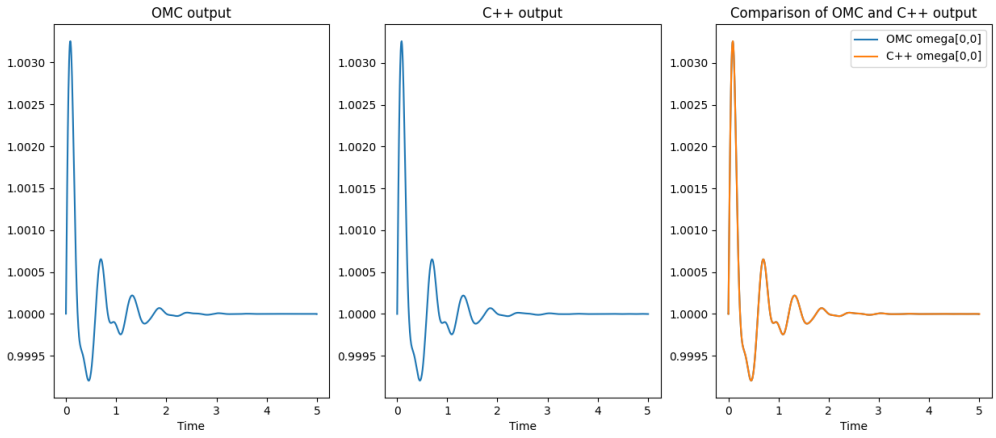

Figure 4.2: Comparison of the variable $omega[1,1]$ values.

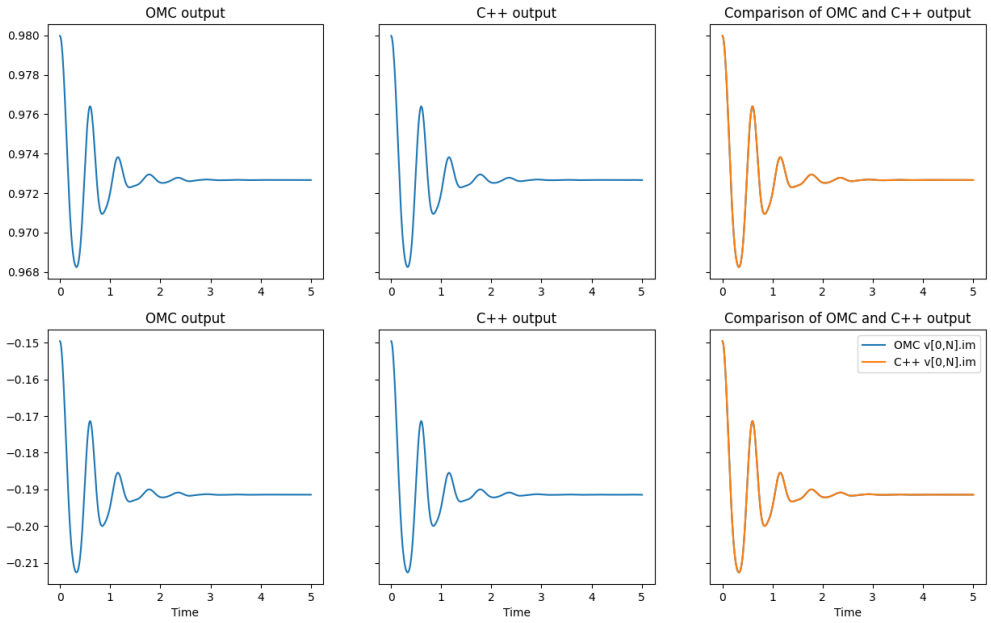

Figure 4.3: Comparison of the variable $v[1, 8]$ (real and imaginary part) values.

Figure 4.2 shows the values of variable $omega[1, 1]$. Figure 4.3 shows the imaginary and real part of the $v[1, 8]$ values. The initial conditions have been ignored for this analysis.

4.2. Scalability of the model

The experiments have been automatised through bash scripts.

The size of the model in the C++ code can be passed as an argument. Listing 18 shows a bash line that runs the model with several inputs. The results are stored in the file `resultsNe.csv` and the output of the run is directed to the file `statsNe.txt`.

The computation is repeated several times (in this case, five times) and the arithmetic mean is used for the complexities and further evaluations. Repeating the simulations more than once will bring less noise in the data, especially for the simulation of the smaller models. In fact, repeating the simulation of the big models doesn't bring much of an advantage in this sense. Hence, for models with thirty-two or more generators per row, the simulation can be executed and measured only once or twice. The fluctuations due to noise won't be as impacting the overall time, in percentage, as much as it is the case for smaller models.

Listing 19 Bash script for the automatisation of the experiments with OMC compiler

```
#!/bin/bash
ulimit -s unlimited

perl omcwrapper.pl 8 >> simtimes_8.txt
mv PowerGridDAE.GridBase_res.csv omc8.csv
mv PowerGridDAE.GridBase omc8
```

Listing 18 Bash script for the automatisation of the experiments with the output of the compilation of the hand-written C++ code

```
#!/bin/bash
ulimit -s unlimited

for (( i = 1; i < 5; i++ ))
do
    echo "iteration $i"
    for ne in 1 2 4 8 11 16 23 32 45 64 90 128
    do
        echo "test $ne"
        ./simulation omega$ne.csv $ne >> stats$ne.txt
    done
done
```

To automatise the testing of PowerGrid.mo with OpenModelica, a Perl script was used. The main operation to perform is to change a line of the source code and overwrite the definition of the parameter N_e. A bash script is then used to perform the automatic compilation and simulation of the model. Listing 19 shows part of the bash script for the OMC compiler. More precisely, it's the set of lines that simulates the model with 8 generators per line.

The Perl script, other than setting the environment for compilation and execution, prints the output times in a file. Those are total time, simulation time, solver time of simulation, compilation time, and the total time of the overall process (time to solution). The script also allows to avoid printing all the variables but selects only four angular speeds `omega_out` and four tensions `v_out` (imaginary and real part) to print in the file CSV. This mimics the behaviour of real-life large-scale model simulations, where only a minority of the variables, of some specific interest, will need to be recorded for later analysis,

while it doesn't make sense to save all the variables of the model; that would generate huge and pretty useless data sets.

4.3. Time complexity

Time complexity will be analysed for both compilation and simulation. The experiments have been run on a server running Linux with Xeon 2650 processor with 20 virtual cores and 72 GB of RAM.

For the compilation time, the OMC data has been retrieved using the OpenModelica tool. At the end of the simulation, OMC prints a set of times. Among those, of particular interest are the total time, simulation time, solver time. Among others, the front-end, back-end, compilation times are printed.

The analysis of both compilation and simulation time is done separately, and the overall times are also compared in this section. The total time from the Modelica file to the results stored for further use includes much more than simply the simulation time (including, but not limited to, pre-processing, compilation, print of the results...). Also, it will be shown later how the size of the arrays greatly impacts the time needed for compilation.

Table 4.1: Times to solution in *seconds*.

N_e	N. equations	OMC Time (s)	C++ Time (s)
1	32	0.741	1.701
2	144	1.32	1.714
4	608	4.08	1.72
8	2.5K	13.2	1.8
11	4.8K	23.1	1.91
16	10K	50.8	2.17
23	21K	114	2.76
32	40K	237	5.13
45	80K	515	10.4
64	163K	1225	24.5
90	324K	3246	65.4
128	640K		231
190	1.4M		882

The total time from Modelica file to results is shown in table 4.1. The data regarding

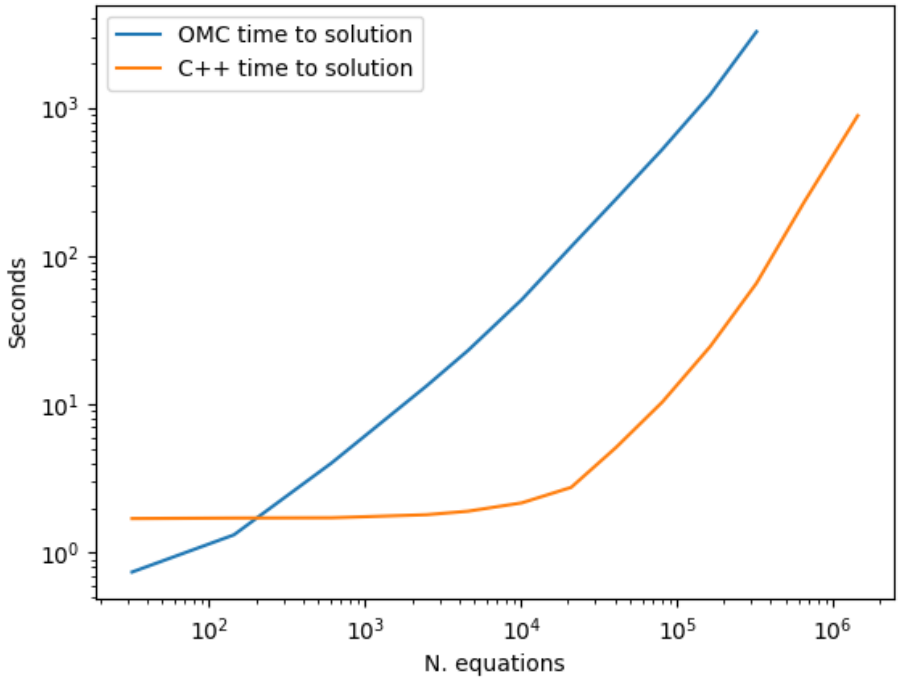

Figure 4.4: Time to solution of the OpenModelica and the hand-written C++ code.

the OMC time to solution is taken from the OMC logs. The C++ data consist of the simulation time which had been added to the compilation time (even though the file was compiled only once and used with all the models). Figure 4.4 shows a graphic comparison of the two complexities. It can be noted how the OMC time to solution is lower than the C++ code time to solution for models smaller than a hundred variables. It will be shown how that is due to the compilation time of the C++ code, constant and around 1.7 seconds. For larger models, the compilation time becomes negligible in comparison with the simulation time of the C++ code. On the other hand, OMC compilation time grows linearly with the number of variables, taking a good part of the time to solution.

4.3.1. Compilation time

The compilation time of the C++ file is around $1.7s$. This time includes the compilation and linking of the PowerGrid simulation file in the HiPerMod project. It is constant since the size of the model is an input argument. It was obtained by running the bash command: `time make -j 20 simulation`.

Table 4.2: Compilation times in *seconds*.

OpenModelica compilation time		
N_e	N. equations	Time
1	32	0.614
2	144	0.979
4	608	2.63
8	2.5K	7.27
11	4.8K	11.1
16	10K	24.3
23	21K	56.8
32	40K	120
45	80K	275
64	163K	691
90	323K	2018
C++ compilation time		
any		1.71

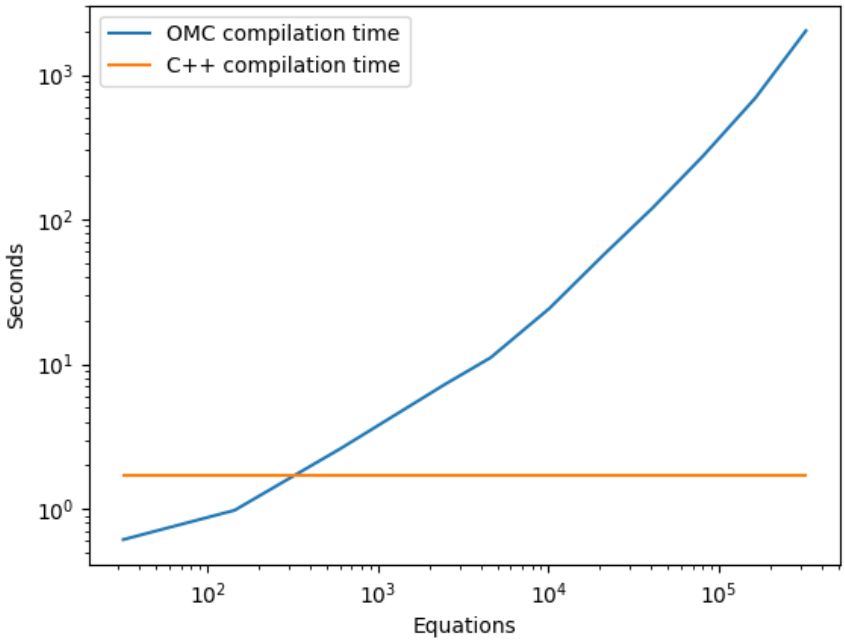

Figure 4.5: Time complexity of the OpenModelica and the hand-written C++ code compilation.

The compilation time of the C++ code refers to the compilation of the benchmark framework, which includes the methods used for the measurements and the print of the results. Those methods come from common libraries, shared with the other benchmark tests. Other than for the measurements, they provide a common framework to standardise the use of the test cases present in the benchmark suite.

The OMC compilation time was taken from the tool's output prints. For every size of the model, a new executable is created. Table 4.2 shows the times w.r.t. the number of equations of each model. Figure 4.5 shows how the OMC tool compilation time crosses the C++ time from models with around two hundred equations. It can be noticed how the OpenModelica compiler has good performances with small models, but it degrades rapidly. The compilation of the test with 90 generators per row (323 thousand equations) has been attempted but it failed after two hours and a half, likely due to memory issues.

The PowerGrid model has a fixed number of vector variables (four complex types and four real type variables). Increasing the size of the model doesn't change the number of for loops and stand-alone equations, hence the C++ constant compilation time.

OpenModelica, on the other hand, flattens the vectors and unrolls the for-loops, producing an increasing number of scalar equations. Every equation is translated into a function (with its arguments and eventual local variables) which is called to compute the residual function. The Jacobian function is also made of several functions, likely one for each variable or equation (translating into the code computing a column or a row of the Jacobian matrix).

Keeping the for-loops and avoiding the creation of a set of identical functions with very little difference from each other (namely, the parameters and variables they use to compute a result) improves the compilation time and the size of the executable.

4.3.2. Simulation time

Simulation times are shown in figure 4.6 and in table 4.3. The C++ code seems to offer a slightly faster computation. However, both in OMC and the C++ code, most of the time is spent in the IDA solver. The residual and Jacobian functions do not play a big part. The Jacobian function is called around a dozen times to simulate the models, while the residual function has between three and four hundred calls. Independently from the model size, the number of calls of the user-provided functions does not increase. Optimising them further wouldn't bring much of an advantage.

Table 4.3: Simulation times in *seconds*.

N_e	N. equations	OMC Time (s)	C++ Time (s)
1	32	0.00125	0.000846
2	144	0.0161	0.0112
4	608	0.0633	0.0212
8	2.5K	0.257	0.105
11	4.8K	0.434	0.206
16	10K	1.43	0.469
23	21K	3.11	1.06
32	40K	8.9	3.43
45	80K	21.8	8.68
64	163K	63.3	22.8
90	324K	171	63.7
128	640K		229
190	1.44M		880

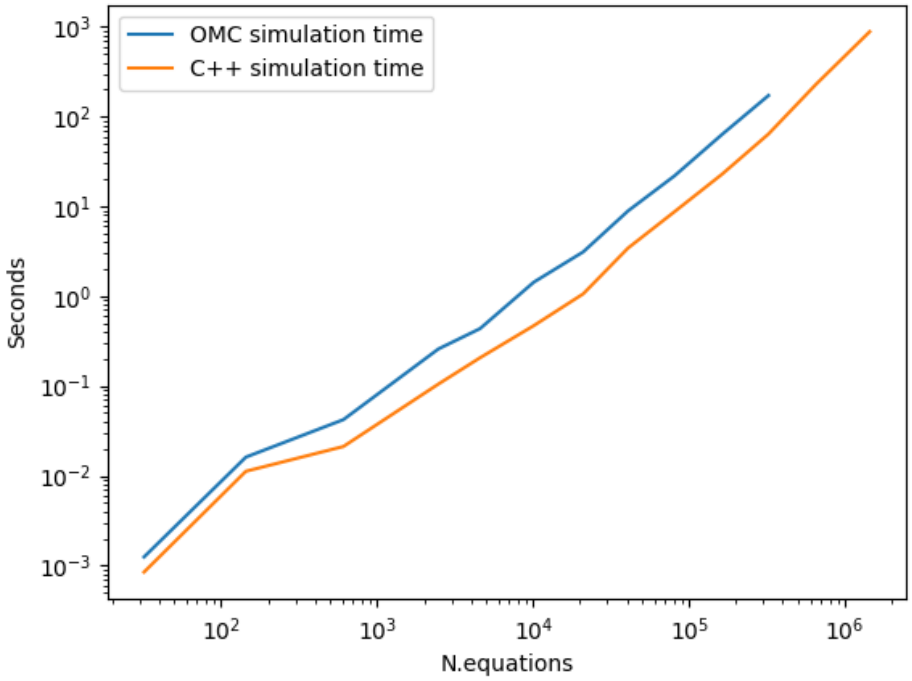

Figure 4.6: Time complexity of the OpenModelica and the hand-written C++ code simulation.

The C++ code allows, among the other statistics, to print the number of calls of and the time spent in the residual and Jacobian functions. Table 4.4 shows the values obtained

Table 4.4: Time (in seconds) and number of calls of the residual and Jacobian functions in the C++ code.

N_e	N.equations	Residual time	Residual calls	Jacobian time	Jacobian calls
1	32	6.209e-06	19	7.817e-06	11
2	144	0.00054311	416	5.5379e-05	20
4	608	0.000761797	309	0.00011919	12
8	2.5K	0.00345356	414	0.000212127	8
11	4.8K	0.00521055	388	0.000448213	10
16	10K	0.00957323	385	0.000750701	10
23	21K	0.0201079	391	0.000965616	7
32	40K	0.0412787	410	0.00193762	7
45	80K	0.0809068	400	0.00383922	7
64	163K	0.144294	383	0.00809147	7
90	324K	0.316378	370	0.0185128	7
128	640K	0.813168	367	0.0641066	10
190	1.3M	2.13611	334	0.144294	10

from the experiments. As can be seen, the number of calls does not depend on the size of the model. Also, the time complexity of both functions is linear w.r.t. the number of equations. This is not surprising, given that the residual function computes exactly as much values as there are equations and the Jacobian function computes at most five times the number of equations values.

The times used by the functions are at most a couple of seconds for the largest model. Further optimisations of the residual function or Jacobian wouldn't bring much advantage to the overall time. Also, it can be noticed how the residual function, even though it is faster, is called hundred of times and takes more time than the Jacobian function. If a future Modelica compiler could use the structure of the array to exploit data locality and lower compilation time, it might not need to optimise the user-provided functions more than what is convenient.

4.4. Space complexity

Table 4.5 shows the sizes of the executable files produced by OpenModelica and by the hand-written optimised C++ code.

OpenModelica produces a different executable file for each value of the parameter N_e, while the C++ code takes N_e as a parameter after compilation. This allows avoiding compiling the same model with a different number of nodes more times. The constant size of the C++ code would be kept even if N_e wasn't an argument of the program but

an internal parameter. It would require changing the code to recompile the new model, but the size would not change (except in the case of compiler optimisations). In fact, the C++ code makes use of vectors as much as possible. The number of vectorial variables doesn't change from one version to the other. Only the size of the variable does, but that impacts the run-time, not the compilation and size of the output.

The values in table 4.5 are computed using the command `size`. The command also shows the BSS and Data segments size (other than the size of the file in hexadecimal representation). BSS and Data segments are constant for all the OpenModelica executable outputs. The size of BSS is 8 bytes, and the size of Data is 2200 bytes. The size of the BSS segment in the C++ file is 568 bytes, and the Data segment is 1864 bytes.

The hand-written C++ code hence allows a constant space complexity, while the OpenModelica generated files grow linearly with the number of equations.

This difference is due to the flattening done in OpenModelica, which transforms a single vector of variables of the same nature into a set of scalar variables. A constant number of vectorial equations (the for loops) are transformed into scalar equations whose amount depends on the size of the vector. The user-provided functions IDA uses, the residual and the Jacobian function, have different sizes depending on the number of variables. The compilation using OpenModelica produces a number of different files. Among those, the user-provided functions had their own. The size of these files might influence the size of the executable. To prove if it's the case, we compute the size of the files containing the Jacobian and the residual function.

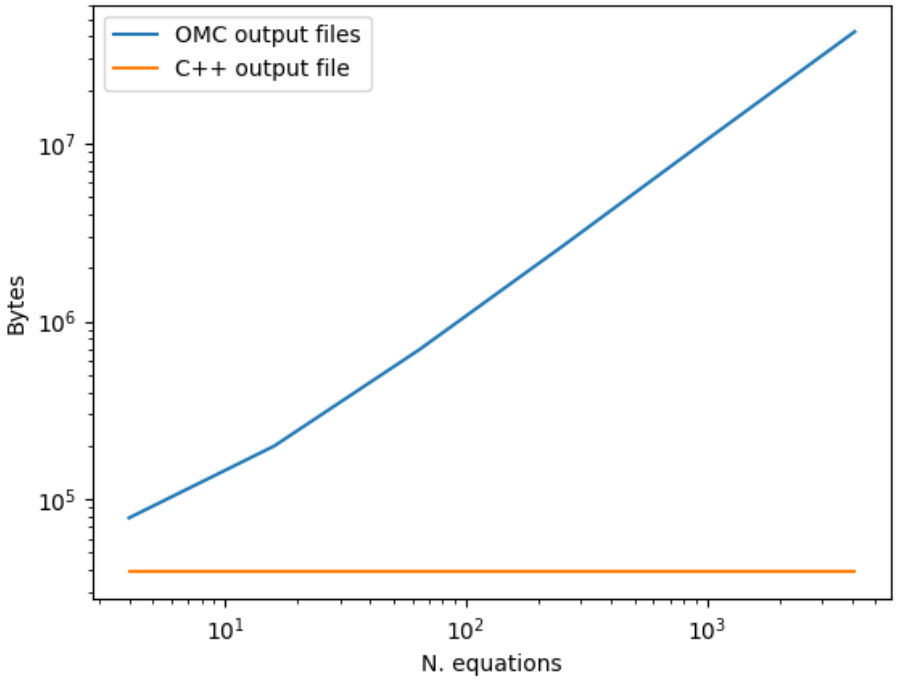

Figure 4.7: Space complexity of the OpenModelica output executable file and of the compiled hand-written C++ code.

Table 4.5: Size of binary executable files

OpenModelica output files			
N_e	N. equations	Total size	Text size
1	32	76.6KB	74.4KB
2	144	192KB	190KB
4	$\sim$ 600	662KB	660KB
8	$\sim$ 2.5K	2.51MB	2.5MB
11	$\sim$ 5K	4.71MB	4.7MB
16	$\sim$ 10K	10MB	10MB
23	$\sim$ 21K	20.6MB	20.6MB
32	$\sim$ 41K	39.9MB	39.9MB
45	$\sim$ 81K	78.9MB	78.9MB
C++ output file			
N_e		Total size	Text size
any		41771	39339

The residual function computes one value for each equation. One expects to notice an increase that is linear with the number of equations.

Table 4.6: Size of *residual function* executable and C files

OpenModelica output files			
N_e	N. equations	File .o size	File .c size
1	32	6965	31KB
2	144	27.8KB	126KB
4	$\sim$ 600	115.6KB	529KB
8	$\sim$ 2500	474.2KB	2.2MB
11	$\sim$ 5K	903KB	4.2MB
16	$\sim$ 10K	1.9MB	9MB
23	$\sim$ 21K	3.9MB	19MB
32	$\sim$ 41K	7.5MB	37MB

Table 4.6 shows the size of the C files containing the residual function(s) and the size of the object file after computation.

Opening one of the files, one can notice how the residual function is in fact a set of smaller functions computing a single residual value. Listing 20 shows the computation of the residual of an equation (a current balance equation, in this case). The function has its header, arguments and executes a single line of code. There are several current balance equations identical to the one presented, and they all have their function.

Listing 21 shows part of the actual residual function. It's a sequence of calls to the smaller functions. Each function call requires building the function environment and then destroying it, which brings worse performance during simulation.

The Jacobian function computes the partial derivatives with respect to every variable for every equation. In the worst-case scenario, in a dense matrix, the space complexity of the data structure is N^2. The PowerGrid model, however, uses a sparse matrix. There is a constant c such that the number of non-zero elements is less than the number of rows (columns) times c:

$$nnz = c * N \ll N^2$$

The size complexity of the Jacobian function is expected to be linear as well.

Listing 20 Residual computation of equation 1619 (numbers go from 1614 to 2257) of the model with 4 generators per row and around 600 equations.

```
/*
equation index: 1619
type: SIMPLE_ASSIGN
i_n[8,6].re = i_v[8,5].re - (i_v[8,6].re - i_h[7,6].re)
*/
void PowerGridDAE_GridBase_eqFunction_1619(DATA *data,
   threadData_t *threadData)
{
  TRACE_PUSH
  const int equationIndexes[2] = {1,1619};
  data->localData[0]->realVars[429]
        /* i_n[8,6].re variable */ =
    data->localData[0]->realVars[541]
        /* i_v[8,5].re variable */
    - (data->localData[0]->realVars[542]
        /* i_v[8,6].re variable */
    - data->localData[0]->realVars[301]
        /* i_h[7,6].re variable */
    );
  TRACE_POP
}
```

Listing 21 Residual function of the model with 4 generators per row and around 600 equations.

```
/* for residuals DAE variables */
OMC_DISABLE_OPT
int PowerGridDAE_GridBase_evaluateDAEResiduals(DATA *data,
   threadData_t *threadData, int currentEvalStage)
{
    //...
  evalStages = 0+1+2+8;
  if ((evalStages & currentEvalStage) &&
        !((currentEvalStage!=EVAL_DISCRETE)?(0):0))
    PowerGridDAE_GridBase_eqFunction_1618(data, threadData);
  threadData->lastEquationSolved = 1618;
    //...
}
```

OpenModelica output files			
N_e	N. equations	File .o size	File .c size
1	32	3781	20.6KB
2	144	15.8KB	90.7KB
4	$\sim$ 600	63.2KB	388.8KB
8	$\sim$ 2500	253.1KB	1.6MB
11	$\sim$ 5K	478.3KB	3.1MB
16	$\sim$ 10K	1MB	6.7MB
23	$\sim$ 21K	2MB	14MB
32	$\sim$ 41K	4MB	27.4MB

Table 4.7: Size of *Jacobian function* executable and C files

Table 4.7 shows the sizes of the C files containing the Jacobian function code and the sizes of the executable files after compilation. As it can be noticed, the size does grow linearly with the number of equations.

Listing 22 shows the computation of a single value of the Jacobian matrix. Like with the residual function, the computation is done in a function. Listing 23 is the function calling several other functions to compute the column of the Jacobian matrix. Computing by column means computing for every variable instead of for every equation. This approach allows, when the Jacobian is not symbolic, to compute a numerical Jacobian through perturbation of variables. Changing a single variable allows us to estimate the derivatives using the effect the perturbation had on the equations. OpenModelica also uses colouring to define groups of variables that can be perturbated at the same time. If two variables do not appear in the same equations, in fact, their effect can't be summed and the numerical computation of the Jacobian can be faster.

Listing 22 Jacobian computation number 423 (numbers go from 233 to 1559) of the model with 4 generators per row and around 600 equations.

```
/*
equation index: 423
type: SIMPLE_ASSIGN
$res_NLSJac5_2.$pDERNLSJac5.dummyVarNLSJac5 = i_n[7,6].re.
   SeedNLSJac5 + i_h[7,6].re.SeedNLSJac5 + i_v[7,6].re.
   SeedNLSJac5 + (-i_h[6,6].re.SeedNLSJac5) - i_v[7,5].re.
   SeedNLSJac5
*/
void PowerGridDAE_GridBase_eqFunction_423(DATA *data,
   threadData_t *threadData, ANALYTIC_JACOBIAN *jacobian,
   ANALYTIC_JACOBIAN *parentJacobian)
{
  TRACE_PUSH
  const int clockIndex = 0;
  const int equationIndexes[2] = {1,423};
  jacobian->resultVars[1] /* $res_NLSJac5_2.$pDERNLSJac5.
     dummyVarNLSJac5 JACOBIAN_VAR */ = jacobian->seedVars[3]
      /* i_n[7,6].re.SeedNLSJac5 SEED_VAR */ + jacobian->
     seedVars[7] /* i_h[7,6].re.SeedNLSJac5 SEED_VAR */ +
     jacobian->seedVars[10] /* i_v[7,6].re.SeedNLSJac5
     SEED_VAR */ + (-jacobian->seedVars[11] /* i_h[6,6].re.
     SeedNLSJac5 SEED_VAR */) - jacobian->seedVars[8] /* i_v
     [7,5].re.SeedNLSJac5 SEED_VAR */;
  TRACE_POP
}
```

Listing 23 Jacobian function computing the values of the 5th column

```
int PowerGridDAE_GridBase_functionJacNLSJac5_column(void*
    inData, threadData_t *threadData, ANALYTIC_JACOBIAN *
    jacobian, ANALYTIC_JACOBIAN *parentJacobian)
{
  TRACE_PUSH

  DATA* data = ((DATA*)inData);
  int index = PowerGridDAE_GridBase_INDEX_JAC_NLSJac5;
  PowerGridDAE_GridBase_eqFunction_422(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_423(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_424(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_425(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_426(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_427(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_428(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_429(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_430(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_431(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_432(data, threadData,
     jacobian, parentJacobian);
  PowerGridDAE_GridBase_eqFunction_433(data, threadData,
     jacobian, parentJacobian);
  TRACE_POP
  return 0;
}
```

An improvement for space complexity for the generated simulation of a Modelica compiler would come from using vectors in the code generation phase instead of flattening all the variables in the pre-processing stage.

Another problem is the use of a single function for every equation, no matter how many times the same equations with different parameters or variables appears. In the PowerGrid model, the current balance equations of the central part of the grid (excluding the borders) are the same equation with a different index (referring to a different node), and the same equation is repeated for the imaginary and the real part. Among the hundreds of residual functions defined, re-using the same for the current balance equations would have certainly impacted the memory needed to store the C file of the residual function. The same reasoning is valid for the computation of the Jacobian. Even if not affecting the number of operations of the simulation, such an improvement would bring an advantage: a shorter C code is faster to compile, and the linking process might be impacted as well by a decrease in the number of symbols to be interpreted. Moreover, every step of the simulation requires to execute the Jacobian and residual functions. The time to load the code into the cache impacts the simulation time.

5 | Conclusions and future developments

In this document, we have presented a benchmark framework for Modelica tools, with a particular interest in Modelica compilers. We have implemented the C++ code for the PowerGrid model described in chapter 2 and 3, and we have analysed the results using the features of the HiPerMod benchmark. The benchmark enables the comparison of existing Modelica tools with hand-written code, and the PowerGrid model provides a test case for models with non-linear DAE equations using the IDA solver. It permits studying the scalability of the tools by setting a single parameter for the test (N_e).

We have compared the hand-written code with the leading open-source software on the market, OpenModelica. The results show how preserving the for-loops and data's structure has reduced the time for both compilation and simulation. The compilation time has become constant w.r.t. the size of the array variables, while it scales linearly with the size of the vectors in OpenModelica. The simulation time speedup of the C++ code isn't as evident as for the compilation time, and it is most likely due to better use of data locality. However, most of the time is spent by the solver IDA. The residual and Jacobian functions require only a small part of the time, making any further optimisation not very effective.

Finally, analysing the residual and Jacobian functions of the code compiled by OpenModelica, we have noticed how every equation is mapped to a function, leading to longer code and compilation time.

5.1. Future work

The poor compilation time taken by OpenModelica makes the simulation of large models unfeasible. The need for large amounts of memory during compilation is also another issue, causing the failure of tests with the largest models. Preserving loops and arrays would transform the complexity of the compilation time from linear (or more than linear) to

constant. Future work related to Modelica compilers includes the integration of IDA into the current version of MARCO. Working with vector equations instead of scalar equations might bring a great improvement in the compilation time and better performance in the initial analysis of the model.

Regarding the HiPerMod benchmark, the next step is to include new test cases of interest to create a benchmark suite covering most if not all of the Modelica language. The final version of the benchmark suite will cover models solvable with both implicit and explicit methods, with difficult and easy residual functions, single and multi-rate. An example of test case yet to be implemented is a model with an almost-fully sparse residual function (i.e. a residual function with a sparse Jacobian matrix, with a fixed number of partial derivatives appearing in each row except for a few rows with a non-fixed number of partial derivatives, growing with the size of the model).

Bibliography

[1] G. Agosta, E. Baldino, F. Casella, S. Cherubin, A. Leva, and F. Terraneo. Towards a High-Performance Modelica Compiler. In *Proceedings of the 13th International Modelica Conference*. Modelica Association and Linkoeping University Electronic Press, Mar. 2019. doi: 10.3384/ecp19157313.

[2] G. Agosta, F. Casella, S. Cherubin, A. Leva, and F. Terraneo. Towards a benchmark suite for high-performance Modelica compilers. In *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '19, page 21–24, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450377133. doi: 10.1145/3365984.3365988. URL `https://doi.org/10.1145/3365984.3365988`.

[3] C. Andersson, J. Åkesson, and C. Führer. PyFMI: A Python package for simulation of coupled dynamic models with the Functional Mock-up Interface. *Technical Report in Mathematical Sciences*, LUTFNA-5008-2016(2), 2016.

[4] M. Arzt, V. Waurich, and J. Wensch. Towards utilizing repeating structures for constant time compilation of large Modelica models. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '14, page 35–38, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329538. doi: 10.1145/2666202.2666207. URL `https://doi.org/10.1145/2666202.2666207`.

[5] K. E. Atkinson. An introduction to numerical analysis. *New York*, 528:38, 1989.

[6] D. C. Augustin, M. S. Fineberg, B. B. Johnson, R. N. Linebarger, F. J. Sansom, and J. C. Strauss. The SCI continuous system simulation language(CSSL). 1968.

[7] E. Baldino. Structural pitfalls of state-of-the-art modelica compilers: An explorative analysis. Master's thesis, Politecnico di Milano, 2018.

[8] F. Bergero, A. Ranade, and F. Casella. QSS and multi-rate simulation of object-oriented models. In *Proceedings of the 7th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '16, page 69–77, New York,

NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342025. doi: 10.1145/2904081.2904091. URL `https://doi.org/10.1145/2904081.2904091`.

[9] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[10] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Consistent initial condition calculation for differential-algebraic systems. *SIAM Journal on Scientific Computing*, 19(5):1495–1512, 1998.

[11] J. C. Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

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

[13] C. F. Curtiss and J. O. Hirschfelder. Integration of stiff equations. *Proceedings of the National Academy of Sciences of the United States of America*, 38(3):235, 1952.

[14] H. Elmqvist. *A structured model language for large continuous systems*. PhD thesis, Lund University, 1978.

[15] H. Elmqvist, D. Brück, and M. Otter. Dymola-user's manual. *Dynasim AB, Research Park Ideon, Lund, Sweden*, 1996.

[16] H. Elmqvist, T. Henningsson, and M. Otter. Systems modeling and programming in a unified environment based on Julia. *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications Lecture Notes in Computer Science*, page 198–217, 2016. doi: 10.1007/978-3-319-47169-3_15.

[17] H. Elmqvist, A. Neumayr, and M. Otter. Modia - dynamic modeling and simulation with Julia. In *JuliaCon 2018*, Jan 2018.

[18] M. Fioravanti. M.A.R.C.O.: an experimental high-performance modelica compiler for large scale systems. 2020.

[19] J. Frenkel, C. Schubert, G. Kunze, P. Fritzson, M. Sjölund, and A. Pop. Towards a benchmark suite for modelica compilers: Large models. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, Linköping Electronic Conference Proceedings, pages 143–152. Linköping University Electronic Press, 2011. ISBN 978-91-7393-096-3. doi: 10.3384/ecp11063143.

[20] A. C. Hindmarsh, C. J. Serban, Radu ande Balos, D. J. Gardner, D. R. Reynolds,

and C. S. Woodward. User documentation for IDA v5.8.0 (Sundials 5.8.0). Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), September 2021.

[21] R. J. Holleman. IEEE Standard VHDL Analog and Mixed-Signal Extensions. *Design Automation Standard Committee if the IEEE Computer Society*, 1999.

[22] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004. doi: 10.1109/CGO.2004.1281665.

[23] B. Lie, A. Palanisamy, A. Mengist, L. Buffoni, M. Sjölund, A. Asghar, A. Pop, and P. Fritzson. OMJulia: An OpenModelica API for Julia-Modelica Interaction. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, page 10. Linköping University Electronic Press, Linköpings universitet, 2019.

[24] S. E. Mattsson and H. Elmqvist. Modelica-an international effort to design the next generation modeling language. *IFAC Proceedings Volumes*, 30(4):151–155, 1997.

[25] S. E. Mattsson, H. Elmqvist, and M. Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, 1998. doi: 10.1016/S0967-0661(98)00047-1.

[26] E. E. L. Mitchell and J. S. Gauthier. Advanced continuous simulation language (ACSL). *SIMULATION*, 26(3):72–78, 1976. doi: 10.1177/003754977602600302. URL `https://doi.org/10.1177/003754977602600302`.

[27] Modelica Association. *Modelica—A Unified Object-Oriented Language for Systems Modelling, Language Specification, Version 3.5*. Modelica Association, 2021. URL `http://www.modelica.org`.

[28] L. W. Nagel and D. Pederson. SPICE (simulation program with integrated circuit emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1973/22871.html`.

[29] M. Otter and H. Elmqvist. Transformation of differential algebraic array equations to index one form. In *Proceedings of the 12th International Modelica Conference*. Linköping University Electronic Press, 2017.

[30] P. Piela. Ascend: an object-oriented computer environment for modeling and analysis. 1989.

[31] M. Scuttari. Design and implementation of a modelica compiler with MLIR and LLVM. 2021.

[32] M. Sjölund, M. Gebremedhin, and P. Fritzson. Parallelizing equation-based models for simulation on multi-core platforms by utilizing model structure. In *17th International Workshop on Compilers for Parallel Computing (CPC 2013), Lyon, France, July 3-5, 2013*, 2013.

A | Appendix A

```
package PowerGridDAE
  model GridBase "Base model of rectangular NxM grid"
    import CM = PowerGridDAE.ComplexMath;
    type PU = Real(final unit = "1");
    type Angle = Real(final unit = "rad");
    type AngularVelocity = Real(final unit = "rad/s");
    type Frequency = Real (final unit = "Hz");
    type Time = Real(final unit = "s");
    operator record ComplexPU = Complex(re(final unit = "1"), im(final unit = "1"));
    constant ComplexPU j = Complex(0,1);
    parameter Integer Ne = 4 "Number of even rows and columns of the grid";
    parameter PU P = 1 "Load active power";
    parameter PU R = 1 "Load nominal resistance";
    parameter PU V = 1 "Load nominal voltage";
    parameter PU X = 0.3 "Line reactance";
    parameter Time Ta = 5 "Time constant of turbogenerators";
    parameter PU sigma = 0.05 "Droop";
    parameter Frequency f_n = 50 "Nominal grid frequency";
    constant PU pi = 3.141592653589793;
    final parameter Integer N = 2*Ne "Number of even rows and columns of the grid";
    final parameter Integer Ng = Ne "Number of generators on each row and column";
    final parameter ComplexPU Y = 1/(j*X) "Line admittance";
    final parameter PU Vg = sqrt(P)*sqrt(1 + X^2/(16*R^2)) "Generator voltage to
    ↪  achieve |Vl| = 1";
    final parameter ComplexPU Vl = Complex(Vg)/Complex(1, (X/4)/4*R)*Vg;
    final parameter AngularVelocity omega_n = 2*pi*f_n;

    ComplexPU i_n[N,N](re(start = i_n_start.re), im(start = i_n_start.im)) "Current
    ↪  out of each node";
    ComplexPU v[N,N](re(start = v_start.re), im(start = v_start.im)) "Node voltages";
    ComplexPU i_h[N-1,N] "Horizontal line currents";
    ComplexPU i_v[N,N-1] "Vertical line currents";

    Angle theta[N, Ng](each start = 0, each fixed = true) "Generator angles";
    PU omega[N, Ng](each start = 1, each fixed = true) "Generator angular velocities";
```

```
  PU Pg[N, Ng] "Active generator power outputs";

  PU Pm[N, Ng] = ones(N, Ng) "Mechanical power input request of generators";

  output ComplexPU v_out[4] "Voltage of the four corner nodes of the grid";
  output PU omega_out[4] "Frequency of the four corner nodes of the grid";

  parameter ComplexPU i_n_start[N,N](re(each fixed = false),im(each fixed = false))
  ↪  "Guess values of current out of each node";
  parameter ComplexPU v_start[N,N](re(each fixed = false),im(each fixed = false))
  ↪  "Guess values of node voltages";
initial equation
  assert(mod(N,2) == 0, "N must be even");

initial equation
  for i in 1:N loop
    for j in 1:N loop
      v_start[i,j]   = if mod(i+j, 2) == 0 then Complex(Vg) else Vl;
      i_n_start[i,j] = if mod(i+j, 2) == 0 then -Complex(Vg,0)/Complex(R, X/4) else
      ↪  Vl/R;
    end for;
  end for;

equation
  // Swing equations for generators
  for i in 1:N loop
    for j in 1:Ng loop
      der(theta[i,j]) = (omega[i,j] - 1)*omega_n;
      Ta*omega[i,j]*der(omega[i,j]) = Pm[i,j] - (omega[i,j] - 1)/sigma - Pg[i,j];
    end for;
  end for;

  // Generators in even nodes, odd rows
  for i in 1:2:N loop
    for j in 1:Ng loop
      v[i,2*j-1] = CM.fromPolar(Vg, theta[i,j]);
      CM.real(v[i,2*j-1]*CM.conj(i_n[i,2*j-1])) = -Pg[i,j];
    end for;
  end for;

  // Generators in even nodes, even rows
  for i in 2:2:N loop
    for j in 1:Ng loop
      v[i,2*j] = CM.fromPolar(Vg, theta[i,j]);
```

```
    CM.real(v[i,2*j]*CM.conj(i_n[i,2*j])) = -Pg[i,j];
  end for;
end for;

 // PQ loads at odd nodes
for i in 1:N loop
  for j in 1:N loop
    if mod(i+j, 2) == 1 then
      v[i,j]*CM.conj(i_n[i,j]) = Complex(P);
    end if;
  end for;
end for;

// Lines
for i in 1:N - 1 loop
  for j in 1:N loop
    i_h[i,j] = (v[i,j] - v[i+1,j]) * Y "Horizontal lines";
  end for;
end for;
for i in 1:N loop
  for j in 1:N-1 loop
    i_v[i,j] = (v[i,j] - v[i,j+1])*Y "Vertical lines";
  end for;
end for;

// Current balances at corners
i_n[1, 1] + i_h[1, 1] + i_v[1, 1] = Complex(0) "Top left";
i_n[N,1] - i_h[N-1,1] + i_v[N,1] = Complex(0) "Top right";
i_n[1,N] + i_h[1,  N] - i_v[1,N-1] = Complex(0) "Bottom left";
i_n[N,N] - i_h[N-1,N] - i_v[N,N-1] = Complex(0) "Bottom right";

// Current balances at edges
for i in 2:N - 1 loop
  i_n[i, 1] + i_h[i, 1] + i_v[i, 1] - i_h[i - 1, 1] = Complex(0) "Top";
  i_n[i, N] + i_h[i, N] - i_v[i, N - 1] - i_h[i - 1, N] = Complex(0) "Bottom";
end for;
for j in 2:N-1 loop
  i_n[1,j] + i_h[1,  j] + i_v[1,j] - i_v[1,j-1] = Complex(0) "Left";
  i_n[N,j] - i_h[N-1,j] + i_v[N,j] - i_v[N,j-1] = Complex(0) "Right";
end for;

// Current balances at internal nodes
for i in 2:N - 1 loop
  for j in 2:N - 1 loop
```

```
      i_n[i, j] + i_h[i,j] + i_v[i,j] - i_h[i-1, j] - i_v[i,j-1] = Complex(0);
    end for;
  end for;

  v_out[1] = v[1,1];
  v_out[2] = v[1,N];
  v_out[3] = v[N,1];
  v_out[4] = v[N,N];

  omega_out[1] = omega[1,1];
  omega_out[2] = omega[1,Ng];
  omega_out[3] = omega[N,1];
  omega_out[4] = omega[N,Ng];
annotation(experiment(StopTime = 5, Interval = 0.01, StartTime = 0, Tolerance =
↪  1e-06),
           __OpenModelica_commandLineOptions= "-d=initialization --daeMode
           ↪  --tearingMethod=minimalTearing --preOptModules-=resolveLoops",
           __OpenModelica_simulationFlags(lv = "LOG_STATS", noEquidistantTimeGrid =
           ↪  "()", nls = "kinsol"));
end GridBase;

operator record Complex "Complex number with overloaded operators"

  replaceable Real re "Real part of complex number" annotation(Dialog);
  replaceable Real im "Imaginary part of complex number" annotation(Dialog);

  encapsulated operator 'constructor' "Constructor"
    function fromReal "Construct Complex from Real"
      import PowerGridDAE.Complex;
      input Real re "Real part of complex number";
      input Real im=0 "Imaginary part of complex number";
      output Complex result(re=re, im=im) "Complex number";
    algorithm

      annotation(Inline=true);
    end fromReal;
    annotation (Icon(graphics={Rectangle(
            lineColor={200,200,200},
            fillColor={248,248,248},
            fillPattern=FillPattern.HorizontalCylinder,
            extent={{-100,-100},{100,100}},
            radius=25.0), Rectangle(
            lineColor={128,128,128},
            extent={{-100,-100},{100,100}},
```

```
          radius=25.0)}));
end 'constructor';

encapsulated operator function '0' "Zero-element of addition (= Complex(0))"
  import PowerGridDAE.Complex;
  output Complex result "Complex(0)";
algorithm
  result := Complex(0);
  annotation(Inline=true);
end '0';

encapsulated operator '-' "Unary and binary minus"
  function negate "Unary minus (multiply complex number by -1)"
    import PowerGridDAE.Complex;
    input Complex c1 "Complex number";
    output Complex c2 "= -c1";
  algorithm
    c2 := Complex(-c1.re, -c1.im);
    annotation(Inline=true);
  end negate;

  function subtract "Subtract two complex numbers"
    import PowerGridDAE.Complex;
    input Complex c1 "Complex number 1";
    input Complex c2 "Complex number 2";
    output Complex c3 "= c1 - c2";
  algorithm
    c3 := Complex(c1.re - c2.re, c1.im - c2.im);
    annotation(Inline=true);
  end subtract;
  annotation (Icon(coordinateSystem(preserveAspectRatio=false,
  ↪  extent={{-100,-100},
           {100,100}}), graphics={
        Rectangle(
          lineColor={200,200,200},
          fillColor={248,248,248},
          fillPattern=FillPattern.HorizontalCylinder,
          extent={{-100,-100},{100,100}},
          radius=25.0),
        Rectangle(
          lineColor={128,128,128},
          extent={{-100,-100},{100,100}},
          radius=25.0),
        Line(
```

```
          points={{-50,0},{50,0}})));
  end '-';

  encapsulated operator '*' "Multiplication"
    function multiply "Multiply two complex numbers"
      import PowerGridDAE.Complex;
      input Complex c1 "Complex number 1";
      input Complex c2 "Complex number 2";
      output Complex c3 "= c1*c2";
    algorithm
      c3 := Complex(c1.re*c2.re - c1.im*c2.im, c1.re*c2.im + c1.im*c2.re);

    annotation(Inline=true);
    end multiply;

    function scalarProduct "Scalar product c1*c2 of two complex vectors"
      import PowerGridDAE.Complex;
      input Complex c1[:] "Vector of Complex numbers 1";
      input Complex c2[size(c1,1)] "Vector of Complex numbers 2";
      output Complex c3 "= c1*c2";
    algorithm
      c3 :=Complex(0);
      for i in 1:size(c1,1) loop
         c3 :=c3 + c1[i]*c2[i];
         /*
     c3 :=Complex(c3.re + c1[i].re*c2[i].re - c1[i].im*c2[i].im,
                  c3.im + c1[i].re*c2[i].im + c1[i].im*c2[i].re);
     */
      end for;

    annotation(Inline=true);
    end scalarProduct;
    annotation (Icon(coordinateSystem(
          preserveAspectRatio=false,
          extent={{-100,-100},{100,100}}),
          graphics={
          Rectangle(
            lineColor={200,200,200},
            fillColor={248,248,248},
            fillPattern=FillPattern.HorizontalCylinder,
            extent={{-100,-100},{100,100}},
            radius=25.0),
          Rectangle(
            lineColor={128,128,128},
```

```
            extent={{-100,-100},{100,100}},
            radius=25.0),
          Line(
            points={{-42,36},{39,-34}}),
          Line(
            points={{-42,-35},{39,37}}),
          Line(
            points={{-55,1},{52,1}}),
          Line(
            points={{-1.5,55},{-2,-53}})}));
  end '*';

  encapsulated operator function '+' "Add two complex numbers"
    import PowerGridDAE.Complex;
    input Complex c1 "Complex number 1";
    input Complex c2 "Complex number 2";
    output Complex c3 "= c1 + c2";
  algorithm
    c3 := Complex(c1.re + c2.re, c1.im + c2.im);
    annotation(Inline=true);
  end '+';

  encapsulated operator function '/' "Divide two complex numbers"
    import PowerGridDAE.Complex;
    input Complex c1 "Complex number 1";
    input Complex c2 "Complex number 2";
    output Complex c3 "= c1/c2";
  algorithm
    c3 := Complex((+c1.re*c2.re + c1.im*c2.im)/(c2.re*c2.re + c2.im*c2.im),
                  (-c1.re*c2.im + c1.im*c2.re)/(c2.re*c2.re + c2.im*c2.im));
    annotation(Inline=true);
  end '/';

  encapsulated operator function '^' "Complex power of complex number"
    import PowerGridDAE.Complex;
    input Complex c1 "Complex number";
    input Complex c2 "Complex exponent";
    output Complex c3 "= c1^c2";
  protected
    Real lnz=0.5*log(c1.re*c1.re + c1.im*c1.im);
    Real phi=atan2(c1.im, c1.re);
    Real re=lnz*c2.re - phi*c2.im;
    Real im=lnz*c2.im + phi*c2.re;
  algorithm
```

```
    c3 := Complex(exp(re)*cos(im), exp(re)*sin(im));
    annotation(Inline=true);
  end '^';

  encapsulated operator function '=='
    "Test whether two complex numbers are identical"
    import PowerGridDAE.Complex;
    input Complex c1 "Complex number 1";
    input Complex c2 "Complex number 2";
    output Boolean result "c1 == c2";
  algorithm
    result := c1.re == c2.re and c1.im == c2.im;
    annotation(Inline=true);
  end '==';

  encapsulated operator function '<>'
    "Test whether two complex numbers are not identical"
    import PowerGridDAE.Complex;
    input Complex c1 "Complex number 1";
    input Complex c2 "Complex number 2";
    output Boolean result "c1 <> c2";
  algorithm
    result := c1.re <> c2.re or c1.im <> c2.im;
    annotation(Inline=true);
  end '<>';

  encapsulated operator function 'String'
    "Transform Complex number into a String representation"
    import PowerGridDAE.Complex;
    input Complex c
      "Complex number to be transformed in a String representation";
    input String name="j"
      "Name of variable representing sqrt(-1) in the string";
    input Integer significantDigits=6
      "Number of significant digits that are shown";
    output String s="";
  algorithm
    s := String(c.re, significantDigits=significantDigits);
    if c.im <> 0 then
      if c.im > 0 then
        s := s + " + ";
      else
        s := s + " - ";
      end if;
```

```
      s := s + String(abs(c.im), significantDigits=significantDigits) + "*" + name;
    end if;
    annotation(Inline=true);
  end 'String';

annotation (versionBuild=2,
versionDate="2019-01-23",
dateModified = "2019-03-20 12:00:00Z",
revisionId="8f65f621a 2019-03-20 09:22:19 +0100",
conversion(
 noneFromVersion="3.2.2",
 noneFromVersion="3.2.1",
 noneFromVersion="1.0",
 noneFromVersion="1.1"),
    Icon(graphics={Rectangle(
          lineColor={160,160,164},
          fillColor={160,160,164},
          fillPattern=FillPattern.Solid,
          extent={{-100,-100},{100,100}},
          radius=25.0), Text(
          lineColor={255,255,255},
          extent={{-90,-50},{90,50}},
          textString="C")}));

end Complex;

package ComplexMath
  "Library of complex mathematical functions (e.g., sin, cos) and of functions
  ↪  operating on complex vectors and matrices"
  final constant Complex j = Complex(0,1) "Imaginary unit";

  function conj "Conjugate of complex number"
    input Complex c1 "Complex number";
    output Complex c2 "= c1.re - j*c1.im";
  algorithm
    c2 := Complex(c1.re, -c1.im);
    annotation(Inline=true);
  end conj;

  function real "Real part of complex number"
    input Complex c "Complex number";
    output Real r "= c.re";
  algorithm
    r := c.re;
```

```
      annotation(Inline=true);
    end real;

    function fromPolar "Complex from polar representation"
      input Real len "abs of complex";
      input Real phi "arg of complex";
      output Complex c "= len*cos(phi) + j*len*sin(phi)";
    algorithm
      c := Complex(len*cos(phi), len*sin(phi));
      annotation(Inline=true);
    end fromPolar;

  end ComplexMath;
end PowerGridDAE;
```

List of Figures

List of Tables

List of Listings

List of Symbols

Variable	Description	SI unit
N_e	Number of even rows or generators per row/column	-
N	Number of nodes (generators and resistances)	-
N_v	Number of variables in the model	-
$omega[i,j]$	Angular velocity of generator [i,j]	[rad/s]
$theta[i,j]$	Angle of generator [i,j]	[rad]
$Pm[i,j]$	Mechanical power input request of generator [i,j]	[p.u.]
$Pg[i,j]$	Active power output of generator [i,j]	[p.u.]
$i_n[i,j]$	Current out of node [i,j]	[p.u.]
$i_v[i,j]$	Horizontal line current between nodes [i,j] and [i+1,j]	[p.u.]
$i_h[i,j]$	Vertical line current between nodes [i,j] and [i,j+1]	[p.u.]
$v[i,j]$	Voltage of node [i,j]	[p.u.]

Acknowledgements

This thesis is the final effort of a long journey. I wouldn't have made it this far without the help of many people.

First and foremost, I'd like to thank my supervisor, Professor Giovanni Agosta, together with the Modelica working group, for the endless patience and support.

I would like to thank my family for always believing in me, even when it was difficult. My gratitude goes to my mother for her years-long efforts, which allowed me to be here today.

I'd like to thank my friends who supported me during the years. Thank you for bearing with me. Without your presence, my university years wouldn't have been this memorable.