INTERNATIONAL JOURNAL OF
NUMERICAL ANALYSIS AND MODELING
Volume 14, Number 6, Pages 916–934

SPARSE AUTOMATIC DIFFERENTIATION FOR COMPLEX NETWORKS OF DIFFERENTIAL-ALGEBRAIC EQUATIONS USING ABSTRACT ELEMENTARY ALGEBRA

SLAVEN PELEŠ AND STEFAN KLUS

Abstract. Most numerical solvers and libraries nowadays are implemented to use mathematical models created with language-specific built-in data types (e.g. `real` in Fortran or `double` in C) and their respective elementary algebra implementations. However, the built-in elementary algebra typically has limited functionality and often restricts the flexibility of mathematical models and the analysis types that can be applied to those models. To overcome this limitation, a number of domain-specific languages such as gPROMS or Modelica with more feature-rich built-in data types have been proposed. In this paper, we argue that if numerical libraries and solvers are designed to use abstract elementary algebra rather than the language-specific built-in algebra, modern mainstream languages can be as effective as any domain-specific language. We illustrate our ideas using the example of sparse Jacobian matrix computation. We implement an automatic differentiation method that takes advantage of sparse system structures and is straightforward to parallelize in a distributed memory setting. Furthermore, we show that the computational cost scales linearly with the size of the system.

Key words. Sparse automatic differentiation, differential-algebraic equations, abstract elementary algebra.

1. Introduction

Differential-algebraic equations (DAEs) are ubiquitous in systems engineering problems, especially in design applications [5, 25, 27, 11, 12]. Mathematical models in this area are typically heterogeneous and very sparse. Obtaining the sparse Jacobian for such problems is critical for successful solving or preconditioning strategies.

Additional challenges arise from the need to effectively manage the complexity of system engineering models. The model equations are typically not in a single central place, but assembled from component model equations loaded from multiple dynamic libraries. Furthermore, the component model structure can (and often does) change at runtime. For example, when designing a heat exchanger one may want to keep inlet and outlet temperatures constant at operating conditions and optimize the heat exchanger geometry parameters. In the transient simulations, however, the heat exchanger geometry is fixed and temperatures are system variables. The variable and parameter designation is selected by the designer as needed at runtime. The ability to reuse the same model for different types of analyses is required to reduce the cost of the computation deployment as well as the cost of component model verification and validation.

To address these requirements, domain-specific modeling languages based on symbolic code manipulations such as gPROMS [17] and Modelica [18] have been introduced. Tools built around these languages [6, 26, 15, 20, 16] allow engineers to work in a more interactive design environment where they can make modifications of their models at runtime. These tools go beyond Jacobian generation and

Received by the editors March 27, 2017.
2010 *Mathematics Subject Classification.* 68W30, 68U20, 68N19.

perform a number of transformations on the mathematical model, such as causalization, tearing, and index reduction to make subsequent simulations more efficient (see e.g. [3] and references therein). Under the hood, the model encoded in the domain-specific language is processed symbolically and code compatible with the numerical solver is generated and compiled on the fly. Then, such a hard-wired precompiled model is simulated and the result is returned to the user. This approach was pioneered in compiler automatic differentiation tools such as ADIC [2] and OpenAD/F [23].

Symbolic manipulations and compiling automatically generated code on the fly allow one to reuse models coded in a domain-specific language for different types of analyses using different numerical solvers. The downside is that the model needs to be regenerated and recompiled every time the model structure is modified. Scaling up this approach to more complex problems is another challenge as the symbolic preprocessing of model equations may become a bottleneck. In such a framework one needs to support two different parallelization schemes – one for the symbolic manipulations of the model equations and another one for solving these equations numerically. Symbolic transformations are generally nontrivial to parallelize. The more features the domain-specific language offers, the more complex the symbolic processing algorithms become and so does their parallel implementation. At the time of this writing, we are not aware of distributed memory parallel schemes for the symbolic processing of mathematical equations.

Modern object-oriented languages, such as C++, which support operator overloading, template specialization, type traits, and other advanced features allow one to create numerical models that can be reconfigured at runtime. In this paper we argue that the same functionality provided by symbolic preprocessing of the model equations can be implemented by creating custom data types and appropriate libraries in mainstream object-oriented languages. Recently, solver frameworks that use abstract data types were proposed [1]. Those frameworks do not require specific data types to be used, but only specify elementary algebra that the data types have to support. By designing models and solvers to use abstract data types, one can reuse the same models and solvers for multiple analysis types such as forward simulations, optimization, sensitivity analysis, or embedded uncertainty quantification [4]. Switching between these may be accomplished simply by changing (or reconfiguration of) the data type. Furthermore, abstract data types can be used to compute automatically the system connectivity graph which could then be utilized, for instance, to partition the system into smaller subsystems, perform index reduction for differential-algebraic equations, implement tearing algorithms, and many other calculations.

We illustrate this abstract elementary algebra approach by developing a method for sparse automatic Jacobian generation, which is superior to other available methods when applied to systems engineering problems. We show that our method allows for model reconfiguration at runtime and overall better code reuse in scientific applications. Moreover, we show that our approach enables the automatic generation of the dependency graph of a system. The method is straightforward to parallelize in a distributed memory environment.

A similar approach is used in Sacado [19], an automatic differentiation package which is part of the Trilinos library [9]. However, Sacado does not support sparse derivatives – it allocates memory for derivatives with respect to all system variables. For many partial differential equation (PDE) models this is not a significant limitation since the sparsity pattern typically consists of locally dense cells and dense