

Author version

Published as:

Gandarillas, V., Joshy, A. J., Sperry, M. Z., Ivanov, A. K., & Hwang, J. T. (2023). A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis. *Structural and Multidisciplinary Optimization*, 67(5), Article 76. <https://lsdolab.github.io/lsdo-publications/publication/gandarillas2023graph/>

BibTeX for this reference:

<https://lsdolab.github.io/lsdo-publications/publication/gandarillas2023graph/>

```
@article{gandarillas2023graph,  
  author = {Victor Gandarillas and Anugrah Jo Joshy and Mark Z. Sperry and Alexander  
    K. Ivanov and John T. Hwang},  
  title = {A Graph-Based Methodology for Constructing Computational Models that  
    Automates Adjoint-Based Sensitivity Analysis},  
  journal = {Structural and Multidisciplinary Optimization},  
  year = {2023},  
  volume = {67},  
  number = {5},  
  pages = {Article 76}  
}
```

A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis

Victor Gandarillas^{1*}, Anugrah Jo Joshy¹, Mark Z. Sperry¹,
Alexander K. Ivanov¹, John T. Hwang¹

^{1*}Department of Mechanical and Aerospace Engineering, University of California San Diego, 9500 Gilman Dr, La Jolla, 92093, CA, USA.

*Corresponding author(s). E-mail(s): vgandari@eng.ucsd.edu;
Contributing authors: ; ; ;

Abstract

The adjoint method provides an efficient way to compute sensitivities for system models with a large number of inputs. However, implementing the adjoint method requires significant effort that limits its use. The effort is exacerbated in large-scale multidisciplinary design optimization. We propose the adoption of a three-stage compiler as the method for constructing computational models for large-scale multidisciplinary design optimization to enable accurate and efficient adjoint sensitivity analysis. We develop a new modeling language called the Computational System Design Language that provides an appropriate input to the compiler front end that works well with multidisciplinary models. This paper describes the three-stage compiler methodology and the Computational System Design Language. The proposed solution uses a graph representation of the numerical model to automatically generate a computational model that computes adjoint-based sensitivities for use within an optimization framework. For two engineering models, the use of the Computational System Design Language reduces the amount of user code by a factor of approximately two compared to their original implementations, without a measurable increase in computation time. This paper also includes a best-case complexity analysis that is built into the compiler implementation for users to estimate the memory required to evaluate a computational model and its derivatives, regardless of which compiler back end ultimately generates the computational model. Future compiler implementations are expected to approach the theoretical best-case memory cost and improve run time performance for both model evaluation and derivative computation.

1 Introduction

In science and engineering, computational models (software implementations used to simulate the behavior of systems) have long been used to aid analysis, avoiding the high cost of building and running experiments to validate theories and designs. Computational models are typically implemented in one of several general-purpose languages (GPLs); C++, Fortran, and Python are popular choices. GPLs provide a higher level of abstraction so that user code is independent of computer architecture. Some of the algorithmic code may be supplied by software libraries, which provide interfaces to common implementations in components from which users can build a computational model. Although GPLs and software libraries provide building blocks for implementing general programs, they do not directly express domain-specific constructs and abstractions [1].

Domain-specific languages (DSLs) simplify the process of implementing software, providing a level of abstraction appropriate for domain experts to define a model of a system without implementing common algorithms. One class of DSLs is algebraic modeling languages (AMLs) [2, 3], which target solving optimization problems. AMLs hide implementation details of derivative computation, memory allocation, etc., allowing users to describe models defining objective and constraint functions in a more natural way. In order to solve optimization problems, AMLs come with interfaces to pre-built solvers. A number of AMLs [4–10] have been developed with various overlapping features, but do not completely eliminate manual or *ad hoc* processes.

The focus of this paper is to automate such manual and *ad hoc* processes, and also present a new AML that hides the implementation details accordingly. We propose the adoption of a three-stage compiler as the method for constructing computational models for large-scale multidisciplinary design optimization problems to enable accurate and efficient adjoint sensitivity analysis. We develop a new modeling language called the Computational System Design Language that provides an appropriate input to the compiler front end that works well with multidisciplinary models. This paper also includes complexity analyses for evaluating computational models and their derivatives.

The paper is organized as follows. Section 2 presents background and a review of prior work. Section 3 presents the graph-based methodology for constructing computational models and the design of the compiler that implements the new methodology. Section 4 presents model code optimizations applied to the graph representation of Section 3.2. Section 5 presents an analysis of the run time and compile time costs incurred by the compiler that implements the methodology presented in Section 3. Section 6 presents an overview of the new algebraic modeling language (AML), the Computational System Design Language (CSDL) and provides resources for the reader to access CSDL and compiler source code for developing numerical and computational models. Section 7 presents a comparison of lines of code and numerical results from applying the analysis presented in Section 5. Finally, Section 8 presents the conclusions of this work along with future work.

2 Background

Higher levels of abstraction for building computational models can be achieved through software libraries, simulation environments, and modeling languages. Each of these methods hides the algorithm implementation from the user and facilitates code reuse, reducing overhead for model developers. Modeling languages provide the highest level of abstraction, allowing users to operate within the application domain, rather than combining generalized components to fit their needs. There are three ways of classifying languages: by their features, implementation, and the types of problems they target, which are described in detail in the sections that follow.

2.1 Modeling Paradigms

While software libraries provide the basic components for building a computational model, they still require the user to implement a computational model based on a mathematical model. Domain-specific languages (DSLs) offer a higher level of abstraction than libraries because users can develop their models at the level of abstraction of the application domain, rather than combining generalized components to fit their needs. Modeling languages are DSLs used for describing models. Like general-purpose languages (GPLs), modeling languages can be categorized according to their features. Each category constitutes a modeling paradigm.

Two similar taxonomies for modeling paradigms are proposed in [11] and [12]. Both taxonomies define modeling paradigms based on the taxonomy of GPLs and use model design structure to categorize models as opposed to the method of execution. In this paper, the term *numerical model* refers to the discretized mathematical model that describes a system. The term *computational model* refers to the software implementation used to simulate the behavior of the system, and also corresponds to the terms *program* and *formal specification* in [11].

Modeling paradigms can be separated into graph-based and language-based paradigms [12]¹. In graph-based paradigms, the user interacts with a graphical user interface (GUI), and in language-based paradigms, the user defines the model by writing source code in a text file. Within the language-based paradigms, there are declarative and imperative families of paradigms (in [11], declarative paradigms are referred to as constraint paradigms). Figure 1 shows a diagram of how the various language paradigms described subsequently are related. Declarative models are so-named because they correspond to declarative programming languages [12], in which the user declares what the desired result should be (or how known and unknown variables are related) without providing the implementation for how to compute the result. In an imperative paradigm, the user primarily communicates how the program changes state during execution.

The family of imperative paradigms contains the procedural and object-oriented paradigms. In a procedural paradigm, variables are allowed to change state throughout the program. Procedural models are referred to as declarative models in [11]. The object-oriented paradigm associates functions with a shared state. Modeling languages

¹The term “graph-based” here refers to a property of languages, and not a property of the methodology presented in this paper.

like ObjectMath [13], Modelica [14], Pyomo [4, 5] and the *ad hoc* language defined by the MDO framework OpenMDAO [15, 16] use the object-oriented paradigm for hierarchical composition of models. The hierarchical model composition is especially useful when constructing multidisciplinary models that combine multiple distinct models often created by communities from different engineering disciplines.

Two paradigms within the family of declarative paradigms are the functional and equation-based paradigms. In the functional programming paradigm, variables are declared to be the result of a function, but the implementation of that function is hidden. The Unified Form Language (UFL) is an example of a functional modeling language [17]. Equation-based modeling uses equations to define noncausal relationships between variables instead of assignments. Noncausal relationships facilitate code reuse by not constraining which variables are inputs and outputs of a model. Examples of declarative modeling languages that use equations rather than assignments include ObjectMath [13] and Modelica [14].

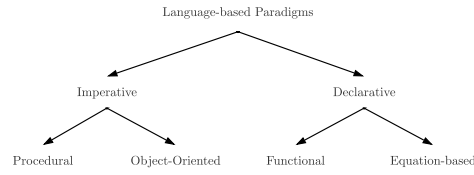


Fig. 1 Diagram showing the relationship between language-based paradigms. Language paradigms can be broadly separated into imperative and declarative paradigms. The imperative family of paradigms contains procedural and object-oriented paradigms, and the declarative paradigm contains the functional and equation-based paradigms.

Modeling languages are not restricted to a single paradigm. For example, ObjectMath and Modelica use the equation-based paradigm to express relationships between variables, and the object-oriented paradigm to express hierarchical model compositions. OpenMDAO and Pyomo also use the object-oriented paradigm for model composition, and the procedural paradigm to assign values to variables. JuMP [6] inherits the multiple paradigms of the Julia language [18], a high performance dynamic programming language with support for automatic differentiation [19, 20] that uses a just-in-time (JIT) compilation approach to generate efficient code. Julia provides a powerful macro system, which JuMP uses to provide users with a simple interface for defining optimization problems. JuMP macros accept problem descriptions in algebraic form, but JuMP also accepts problem descriptions defined in native Julia functions, which can be more verbose than the original model definition, and use a procedural style.

The AML presented in this paper, the Computational System Design Language (CSDL), uses a functional paradigm to express relationships between variables, and an object-oriented paradigm to compose models hierarchically.

2.2 Implementation Strategies

Besides categorizing modeling languages by paradigms, the method of generating an executable object differentiates languages as well. With the exception of Pyomo and OpenMDAO, the modeling languages discussed in Section 2.1 all generate code at a lower level of abstraction, maintaining the readability, reusability, and quality of the code written at the higher-level language and the performance of the lower-level language, e.g. FORTRAN, C, and C++. Pyomo and the OpenMDAO framework, on the other hand, do not generate low-level code, and the Python interpreter evaluates the computational model directly.

Some languages provide a single compiler for generating low-level code, while others are implemented so that multiple compilers or compiler back ends may be used to generate low-level code for the same numerical model. The FEniCS project provides users with the ability to define partial differential equations in weak form using the Unified Form Language (UFL) [17] and compile programs using either the FEniCS Form Compiler (FFC) [21–24] or the SyFi Form Compiler (SFC) [25, 26].

2.3 Algebraic Modeling Languages (AMLs)

Modeling languages are not only classified by their features or how they are implemented; they are also classified by the types of problems they are designed to solve. Algebraic modeling languages (AMLs) are DSLs targeting optimization problems. AMPL and GAMS [7–9], two commercially available AMLs, provide derivatives for solving optimization problems when using a gradient-based approach is appropriate, and both scale well with the number of variables. They do not however, provide a hierarchical model composition to either organize model disciplines or capture coupling between model disciplines. Pyomo [4, 5] is an AML that provides semantics for hierarchical composition of models. Optimica [10], an extension of Modelica, supports dynamic optimization. Both use object-oriented features to specify the topology of physical systems.

Users of Pyomo define relationships between variables using native Python code that is automatically differentiated. Defining relationships between variables, however, is done imperatively, which requires the user to translate equations from a model definition to a lower-level procedure. Finally, all Pyomo simulations are implemented in Python, which prevents users from having access to more efficient low-level implementations.

The OpenMDAO framework is a modeling framework implemented in Python, designed for multidisciplinary design optimization (MDO). OpenMDAO defines specialized Python classes for users to compose models hierarchically. The same concept of a connection as in ObjectMath and Modelica is used to capture the nonlinear coupling between disciplines. OpenMDAO implements the modular analysis and unified derivatives (MAUD) architecture [27], which enables the use of hierarchical solvers to evaluate the model, and automatically solves the unifying derivative equation (UDE) [28] to compute total derivatives, overcoming the curse of dimensionality.

None of the modeling languages presented here enable fully automatic computation of derivatives using adjoint-based sensitivity analysis. Although OpenMDAO automates the computation of total derivatives by solving the unifying derivatives equation (UDE)², users must provide implementations for computing partial derivatives or resort to using a numerical approximation. Combining the capability of computing partial derivatives automatically and computing total derivatives via the adjoint method are key benefits of the graph methodology of this paper.

2.4 A New Approach to Generating Computational Models

The preceding software libraries, modeling languages, and frameworks provide a higher level of abstraction than what is possible with low-level general-purpose languages. Each modeling language or framework provides a subset of the following: hierarchical composition for modeling multidisciplinary systems, declarative style of defining relationships between variables, fully automated computation of derivatives when disciplines are coupled, automatic code generation, and an interface for an optimizer to solve an optimization problem. The Computational System Design Language (CSDL), presented in Section 6, is a functional and object-oriented algebraic modeling language (AML). CSDL is an embedded domain-specific language (EDSL) that provides hierarchical composition (as do OpenMDAO, Pyomo, ObjectMath, and Modelica) for modeling multidisciplinary systems, a functional style of defining relationships between variables (as does UFL), fully automated computation of derivatives when disciplines are coupled (which OpenMDAO achieves in part), automatic code generation (as do AMPL, GAMS, and UFL), and an interface for an optimizer to solve an optimization problem (as do all AMLs).

3 Model Code Generation

This section presents a novel graph-based methodology for building computational models that forms the contribution of the paper. The methodology constructs a graph representation of the numerical model, where the nodes of the graph (Definition 1) represent variables and operations, and each edge represents the dependence of a variable on an operation or an operation on a variable. Using a graph representation provides a standard structure from which a computational model may be generated. The graph representation also enables efficient transformations to the model without changing the original model definition. The methodology is implemented as a three-stage compiler that translates a numerical model to a computational model for use in numerical optimization. Numerical models are defined in an AML such as the Computational System Design Language (CSDL), described in Section 6.

Optimization problems are solved by updating design variables until the objective is minimized and constraints are satisfied. The process of evaluating the objective, constraint, objective gradient, and constraint Jacobian is separate from the process of updating the design variables. For simplicity, the computational model is referred to

²There are two modes when solving the UDE: forward mode and reverse mode. Forward mode corresponds to using the direct method and reverse mode corresponds to using the adjoint method.

as a *Simulator*³, which is the name used in CSDL. Figure 2 shows how a Simulator, responsible for evaluating the computational model, connects to an Optimizer, which is responsible for updating design variables to find a solution to an optimization problem. The Simulator object computes outputs (including the objective and constraints), the objective gradient, and the constraint Jacobian. The design variables, objective gradient and constraint Jacobian, are then transferred to the Optimizer at the end of each iteration. The objective gradient and constraint Jacobian may be computed using any differentiation method.

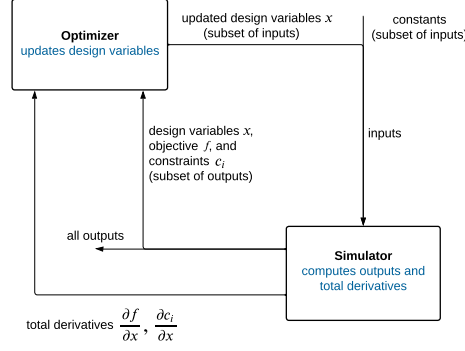


Fig. 2 The compiler constructs a Simulator object that evaluates the computational model within an optimization framework. The Simulator and Optimizer execute sequentially until the Optimizer converges to an optimal solution.

3.1 Compiler Overview

The methodology presented in this paper is formulated as a three-stage compiler that constructs the Simulator object in Figure 2. Figure 3 shows the design of the overall compiler architecture. A three-stage compiler generates an intermediate representation (IR) from source code and then generates an executable object from the IR. General-purpose languages commonly use three-stage compilers to share functionality across compilers for the same language, or for multiple languages to take advantage of the same compiler architecture. One example is the LLVM compiler framework, which provides a back end for many compiled languages [29]. The IR that the compiler in this paper uses represents a user-defined model in the form of a directed acyclic graph (DAG). A DAG is a directed graph (Definition 1) with no cycles (Definition 3).

Definition 1 (Directed Graph). *A directed graph is a pair $G = (\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ is a set whose elements are called nodes, and $\mathcal{E}$ is a set of ordered pairs of nodes (u, v) called edges.*

Definition 2 (Directed Path). *A directed path is a sequence of edges $(e_1, e_2, \dots, e_{n-1})$ for which there is a sequence of nodes $(v_1, v_2, \dots, v_n)$ such that $e_i = (v_i, v_{i+1})$ for $i = 1, 2, \dots, n-1$.*

Definition 3 (Cycle). *A cycle is a directed path (Definition 2) where $v_1 = v_n$.*

³This is the same as the concept of a model within the `Problem` class in OpenMDAO

Definition 4 (Directed Acyclic Graph). A *directed acyclic graph (DAG)* is a directed graph without cycles.

The variables and functions within a system model, plus their dependency relationships can be represented by a DAG (Definition 4), where each node represents a variable or an operation, and each edge represents a dependency relationship between two nodes.

A three-stage compiler generates an intermediate representation (IR) from the source code defining a program, performs operations on the IR to optimize the program, and then generates executable code from the IR. In this work, the source code describes a numerical model, the IR is a DAG, and the generated executable code evaluates a computational model. Figure 3 shows a schematic of the compiler architecture. Each stage of a three-stage compiler can be interpreted as a function mapping one object to another. If $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a numerical model and $\mathcal{M}$ is the set of possible models f , a compiler front end can be interpreted as a function $p : \mathcal{M} \rightarrow \mathcal{D}$ that generates a DAG, $G \in \mathcal{D}$, where $\mathcal{D}$ is the set of all DAGs.

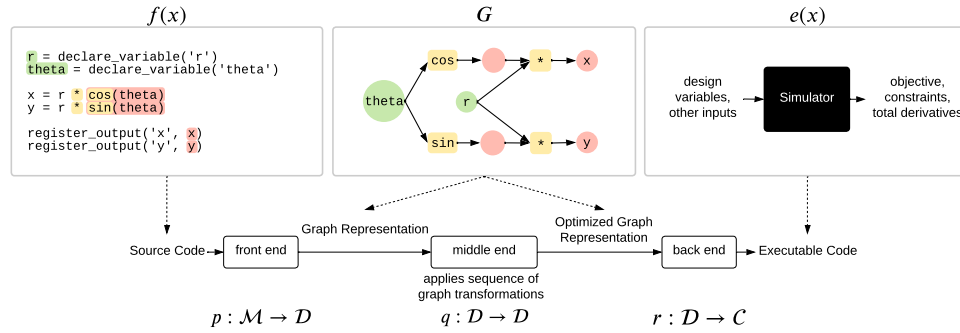


Fig. 3 A three-stage compiler translates a numerical model to executable code implementing the computational model. The user supplies source code and the compiler generates a *Simulator* that evaluates the computational model.

A compiler middle end applies a sequence of graph transformations to the graph representation to optimize the graph representation independent of any strategy used to generate the executable object. A compiler middle end can be interpreted as a function $q : \mathcal{D} \rightarrow \mathcal{D}$ that transforms the graph representation $G \in \mathcal{D}$ of a model $f \in \mathcal{M}$ into another graph representation. Note that a function q may be a composite function consisting of a *sequence* of graph transformations applied to G so that the compiler back end generates efficient code.

A compiler back end can be interpreted as a function $r : \mathcal{D} \rightarrow \mathcal{C}$, where $\mathcal{C}$ is a set of functions representing the set of possible computational models. The compiler back end can also be used for implementing partial derivative calculations for all functions in the standard library for a given AML, which are then used to compute the total derivatives for the system-level model. By automating the computation of partial and total derivatives, the generated computational model automates derivative

computation for complex, high-dimensional, multidisciplinary systems. The compiler back end is free to generate the computational model as a native executable or using any language that can subsequently be compiled to native code.

The algebraic modeling language (AML) that the compiler front end is designed to translate must provide interfaces to vectorized continuous and differentiable functions within a standard library, which the compiler back end is responsible for implementing. Designing the compiler without restricting the generated code to use any particular language removes any limitation to efficient execution imposed by building a new, less mature compiler from scratch for a particular language. Eliminating such a limitation creates the opportunity to improve computational efficiency of the computational model, which is especially important when solving an optimization problem due to the possibly large number of model evaluations required.

Users define a numerical model in an AML, and the compiler constructs a graph representation, followed by a Simulator object according to the user's choice of compiler back end. The back end generates executable code from the (optimized) graph representation. One compiler front end is implemented to translate numerical models from the Computational System Design Language (CSDL) (Section 6), to a graph representation described in Section 3.2.

3.2 Graph Representation of Models

The graph representation is an intermediate representation (IR) with a directed acyclic graph (DAG) structure (Definition 4). Each node represents a multidimensional array variable or vectorized operation, and the edges represent dependency relationships between variables and operations, as shown in Figure 3. Each variable node can depend on at most one operation node, and each operation node has at least one variable node that depends on it. Each operation node represents a function. The DAG structure encodes a partial ordering of all nodes, which is used to determine the execution order of the operations.

The sequence of functions that apply transformations to the graph representation comprise the compiler middle end. These transformations (Section 4) ultimately optimize the computational efficiency of the computational model. The compiler can also gather statistics about the model prior to generating executable code, which can provide users some insight into model properties and implementation trade-offs involved in setting model parameters. The compiler provides an interface for the user to interact with the graph representation, including applying transformations for optimizing the graph representation and visualizing the graph representation and its adjacency matrix prior to generating executable code. This interface provides access to the analyses presented in Section 4 and the capability to perform these analyses, such as those shown in Section 7.2.

The compiler graph representation is similar to that of the IR used in the UFL, which also has a DAG structure, where each node is an expression and each edge is a dependency [17]. Both this compiler graph representation and UFL IR nodes represent array variables and vectorized operations. Doing so reduces the size of the graph as shown in Figure 4.

The compiler graph representation is similar to a computational graph used in artificial neural networks (ANNs) and serves the same purpose as an expression graph used in automatic differentiation (AD). Although the graph representation is similar to a computational graph and an expression graph, there are some differences in representation. The nodes in an AD expression graph represent variables, and the edges encode dependency relationships between the variables.

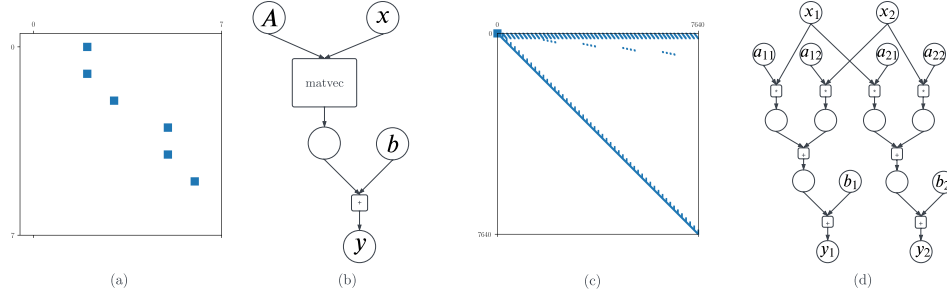


Fig. 4 Adjacency matrices and graph representations of $y = Ax + b$, showing that the size of the variables in a numerical model has no effect on either the size or the order of the graph representation. (a) Adjacency matrix sparsity pattern for graph representations generated by compiler front end using one node, each representing a variable x, y, b, A of any compatible dimensions, (b) the corresponding graph representation, (c) Adjacency matrix sparsity pattern for graph representations generated by compiler front end using a node to represent each scalar value of x, y, b, A , where $x, y, b \in \mathbb{R}^{50}$, $A \in \mathbb{R}^{50 \times 50}$, and the intermediate scalar values and operations involved in Ax . (d) Visualization of a graph with nodes representing scalar values, where $x, y, b \in \mathbb{R}^2$, $A \in \mathbb{R}^{2 \times 2}$.

The graph representation has a unique way of representing variable coupling. Due to the acyclic property of the DAG, *edges* can only represent *explicit* dependencies between variables and operations, and cannot encode coupling between variables. Instead of representing coupling by *edges* that form a cycle, the graph representation uses implicit operation nodes to represent coupling between variables. Each implicit operation node contains a graph representation defined by a model, which defines residual variables explicitly in terms of known and unknown variables. These residual variables must converge to zero at run time using a root-finding solver to compute the unknown variables. Figure 5 shows a schematic of how the graph representation represents an implicit operation as a node containing another graph representation for a model defining a residual for an implicit operation.

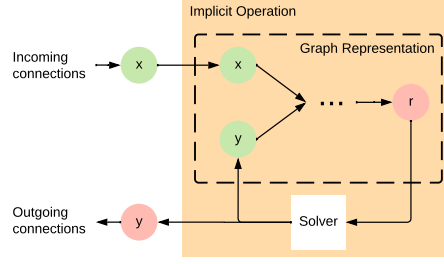


Fig. 5 An implicit operation defined by a model with known variable x , unknown variable y , and residual r . Users provide a model definition, and a choice of a root-finding solver, but do not provide implementation details other than solver options, e.g. maximum number of iterations.

Isolating coupled relationships within specialized nodes preserves the acyclic nature of the DAG, enabling a topological ordering of nodes, which guarantees that operations are executed after their argument values become available. Figure 6 shows the adjacency matrices for a model with two implicit operation nodes.

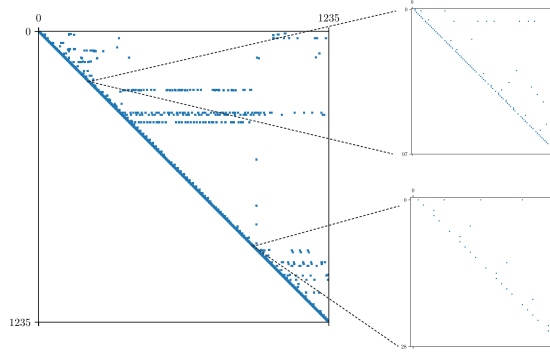


Fig. 6 Adjacency matrix for a model of a propeller rotor using the blade element momentum-based ideal-loading design (BILD) method with two implicit operations, along with an adjacency matrix for each model defining each implicit operation.

3.3 Derivative Computation

In order to apply a gradient-based approach to multidisciplinary design optimization problems, the computational model must compute derivatives. This section presents how a compiler back end automates the implementation of derivatives for a computational model using the graph representation of the numerical model. Multidisciplinary design optimization (MDO) models have particular structure and characteristics that

is lacking in other models (such as neural networks) whose structure can be represented by a graph. For example, operations or collections of operations that are not typically present in an artificial neural network, but are common in MDO problems include those that represent the evaluation of surrogate models and numerical integrators. Nested implicit operations (Section 3.2) also appear more often in MDO models due to coupling between disciplines. A compiler back end uses the graph representation in Section 3.2 to implement the adjoint method to compute total derivatives for a computational model used within an MDO setting.

The graph representation is specifically designed to represent models used in MDO problems and decoupled from how a compiler back end implements derivatives. A compiler back end can be designed to implement derivative computation using any strategy, including automatic, symbolic, and numerical differentiation. The generated computational model can also call user-defined or external code (e.g. a surrogate model or CFD solver) that provides its own derivative implementation⁴. The Computational System Design Language (Section 6) provides custom operations for this purpose.

Although this paper does not present an Automatic Differentiation (AD) library, is worth noting that AD libraries follow a similar approach to our methodology, constructing their own expression graphs, similar to the graph representation in Section 3.2, prior to computing exact derivatives [20, 30]. Just as a compiler back end in the three-stage compiler presented in Section 3.1 operates on the graph representation, Enzyme operates on the LLVM intermediate representation (IR), implementing AD for a wide variety of general purpose languages, including C, C++, FORTRAN, Rust, and Swift [31–33]. Because this graph representation is independent of derivative implementation, an AD library could potentially be implemented to use this graph representation rather than constructing its own, simplifying the implementation of the AD library.

4 Model Code Optimizations

The graph representation in Section 3.2 serves two purposes. The first is to provide a standard representation of a numerical model written in any language for which a compiler front end is developed, and to implement a computational model according to any implementation strategy that a compiler back end may use. The second purpose of the graph representation is to provide the compiler middle end a way to perform implementation-independent optimizations on model code prior to generating a computational model. Implementation-independent optimizations include changes to the graph representation that result in improvements to efficiency in terms of both performance and memory. The optimizations in this section are applied to the graph representation and make no modifications to the user’s source code. Therefore, users are not required to change their numerical model definitions to take advantage of these optimizations, regardless of which language they use to define the numerical model or which compiler back end users choose to generate a computational model.

⁴The derivative implementation for external code may be inexact due to limitations in how the external code was implemented.

4.1 Optimal Ordering of Graph Nodes

The graph representation of the model is a directed acyclic graph (DAG), from which the order of execution may be computed by performing a topological sort. The total ordering on the operation nodes determines the order of execution and ensures an upper triangular adjacency matrix as well as a lower (block) triangular Jacobian of partial derivatives. Kahn's algorithm [34] provides an efficient way to compute a topological ordering. A DAG, however, does not have a unique topological ordering. Although the choice of ordering does not affect implementation efficiency in terms of performance, there is an impact on the memory cost of evaluating a computational model. This section provides a description of computing the optimal ordering for minimizing the memory cost for evaluating a computational model at run time.

The objective is to minimize the maximum amount of memory allocated during program execution. This implies that each variable must be allocated for the shortest amount of time possible; i.e., allocated as late as possible before it is first used, and deallocated as early as possible after it is no longer necessary for continuing to evaluate the computational model. Choosing an ordering of nodes can reduce the maximum amount of memory allocated during program execution. Kahn's algorithm, shown in Algorithm 1, performs a topological sort on a DAG. The algorithm sorts the DAG by visiting the edges according to a depth-first search strategy, visiting first the nodes with zero outdegree stored in a stack S .

Algorithm 1 Kahn's Algorithm

```

1: procedure TOPOLOGICALSORT( $G(\mathcal{N}, \mathcal{E})$ ,  $R = \{r \in \mathcal{V} : \deg^+(r) = 0\}$ )
2:    $L = ()$ 
3:    $S = (r_1, \dots, r_n)$ , where  $r_i \in R, i \in \{1, \dots, n\}, n = |R|$ 
4:   while  $S$  not empty do
5:      $v = \text{pop}(S)$ 
6:      $\text{push}(L, v)$ 
7:     for  $w$  in  $\{w : (v, w) \in \mathcal{E}\}$  do
8:        $\text{remove}(\mathcal{E}, (v, w))$ 
9:       if  $\deg^+(w) = 1$  then
10:         $\text{push}(S, w)$ 
11:       end if
12:     end for
13:   end while
14:   return  $\text{reverse}(L)$ 
15: end procedure

```

By traversing the nodes in S and $\{w : (v, w) \in \mathcal{E}\}$ for each $v \in \mathcal{N}$ in the right order, variable allocation may be delayed as much as possible. A function $u : \mathcal{D} \times \mathcal{N} \rightarrow \mathbb{N}$ that counts the number of nodes for which there exists a directed path [2](#) from those nodes to the current node, including the current node, is used to sort nodes in the list S and sets $\{w : (v, w) \in \mathcal{E}\}$ for each $v \in \mathcal{N}$. If the nodes in S are initially sorted by $u(G, v)$ in descending order, then the nodes initially stored in S appear in L in ascending

order of $u(G, v)$. For each $v \in \mathcal{N}$, if the nodes in $\{w : (v, w) \in \mathcal{E}\}$ are initially sorted by $u(G, w)$ in descending order, then the nodes initially stored in $\{w : (v, w) \in \mathcal{E}\}$ with the same outdegree $\deg^+(w)$ prior to applying Kahn’s algorithm appear in L in ascending order of $u(G, w)$. After reversing L , the result of the topological sort is one that shifts the allocation of variables to the end of the list as close as possible to the point in the program’s execution where those variables are no longer needed for continuing to evaluate the computational model and may be deallocated, thus minimizing the maximum memory cost reached during program execution.

4.2 Dead Code Removal

Not all user code in the numerical model is necessarily translated to the computational model. The Computational System Design Language (CSDL) requires that all outputs that are registered have a name that the user can use to access their run time values. Registered outputs are represented by variable nodes in the graph representation, which stores edges to indicate a dependency relationship between operations that define the outputs (predecessors) and operations that use the outputs as arguments (successors). In some cases, a user does not register an output, indicating that that output is either not an objective or constraint in the design problem, or is not an output that the user wishes to access at run time.

If an output that is not registered has no successors, then some nodes may be removed safely from the directed acyclic graph (DAG) with no effect on the result of evaluating the computational model. The compiler is allowed to remove all nodes that lie only on paths leading to unregistered outputs with zero successor nodes, which is equivalent to removing “dead code”, or code that has no effect on the output. The compiler accomplishes the removal of dead code by modifying the graph input to Kahn’s algorithm. Kahn’s algorithm [34] takes a list of nodes S , including only those nodes with zero successors ($\deg^-(r) = 0$) as input to Algorithm 1. The result is that the condition on line 9 of Algorithm 1 is never met, so any node that does not influence a registered output is excluded from the list of nodes L , and thus the generated executable object.

4.3 Combining Element-wise Operations

Many standard functions are vector-valued functions of the form $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, whose scalar outputs have element-wise dependence on the scalar inputs. Functions that define element-wise relationships have diagonal Jacobians. Mathematical models often contain variables that are defined by a composition of functions whose Jacobians are diagonal. The Jacobian of the resulting composite function is also diagonal. This section shows how the compiler decides when to apply two different approaches of computing partial derivatives for composite functions with diagonal Jacobians, and analyzes the relative computational cost of both approaches.

One approach to computing partial derivatives for a composite function with diagonal Jacobian is to apply the chain rule. Applying the chain rule to a composition of m functions with Jacobians of $n \times n$ dimensions requires computing m partial derivatives, followed by $m - 1$ matrix multiplications. The $m - 1$ matrix multiplications

involve $(m - 1)n$ scalar multiplications. The number of scalar operations required to compute the derivatives of the composite function is given by

$$\sum_{i=1}^m p^{(i)} + (m - 1)n \quad (1)$$

where $p^{(i)}$ is the cost of computing the derivative of the i^{th} function. An alternative approach is to use a numerical approximation to compute the partial derivatives of the composite function, which may be a less computationally expensive approach, depending on the cost of computing partial derivatives $p^{(i)}$.

Two numerical approximation methods—the finite difference method, and the complex step method [35]—perturb each element of the input and then compute the ratio of the change in each output with respect to the size of the perturbation. Computing derivatives using a numerical approximation requires one function evaluation per perturbation, for a total of at least n function evaluations, depending on the approximation used. For functions with diagonal Jacobian, however, computing derivatives by numerical approximation can be accomplished by perturbing all the elements of the input vector at once, requiring only *one* function evaluation when using the complex step method:

$$\frac{\partial f}{\partial x} = \text{diag} \left(\frac{\text{Imag}[f(x + ih\mathbf{1}_n)]}{h} \right) \quad (2)$$

where $h \in (0, 1)$ is the step size. The number of scalar operations required to compute the derivative of a composite function composed of m functions, each of the form $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is given by $mn + 2n$. The mn term accounts for evaluating each of the m functions chained together to form the composite function. The $2n$ term accounts for computing $x + ih\mathbf{1}_n$ and dividing the n elements of $\text{Imag}[f(x + ih\mathbf{1}_n)]$ by h .

In order for the complex step approximation to offer a performance benefit when computing the partial derivatives of the composite function, the time cost of computing analytic derivatives and performing matrix multiplications to apply the chain rule $\sum_{i=1}^m p^{(i)} + (m - 1)n$ must exceed the time cost of applying the complex step approximation to the composite function $mn + 2n$.

Assuming that the time cost of computing the analytic derivatives of any element-wise function is the same as computing the original element-wise function, $p^{(i)} = n, \forall i \in \{1, \dots, m\}$. Then from Equation (1),

$$mn + (m - 1)n \geq mn + 2n \quad (3)$$

$$mn + mn - n \geq mn + 2n \quad (4)$$

$$mn \geq 3n \quad (5)$$

$$m \geq 3 \quad (6)$$

That is, if a function is composed of a chain of three or more element-wise functions, then applying the complex step method to the composite function requires fewer scalar operations than applying the chain rule. If $p^{(i)} > n$, then in some cases, approximating the derivative for a composition of as few as two element-wise operations is less computationally expensive than applying the chain rule.

The compiler middle end traverses the graph representation of the model (Section 3.2) to identify paths containing operation nodes representing only functions with diagonal Jacobians and combines them into a single operation representing the a composite function with diagonal Jacobian.

5 Computational Cost Analyses

This section presents analyses of the time and memory costs associated with generating and evaluating a computational model, using the compiler graph representation presented in Section 3.2. Although implementation details affect the amount of memory required to run a simulation, the compiler can use the graph representation of the model to estimate minimum memory requirements (as measured by the number of scalar values the must be allocated at any time during simulation) for evaluating the computational model. These memory requirements may guide the user to select different model parameters to generate a computational model with different levels of fidelity to accommodate available resources. The compiler front end provides an interface for users to perform the analyses presented in this section for any model prior to generating a computational model. Numerical results demonstrating the potential for improved compile time and run time costs are shown in Section 7.

5.1 Nomenclature

The compiler front end generates a directed acyclic graph (DAG) representing a model, denoted by $G(\mathcal{N}, \mathcal{E})$ (Definition 4), where $\mathcal{N}$ is the set representing of nodes, which is defined by the union of the variables $\mathcal{V} \subset \mathcal{N}$ and operations $\mathcal{O} \subset \mathcal{N}$, and $\mathcal{E} = \{(u, v) : u \in \mathcal{V} \Leftrightarrow v \in \mathcal{O}, u \in \mathcal{O} \Leftrightarrow v \in \mathcal{V}\}$ is the set of directed edges representing dependencies between variables and operations. The graph G stores each variable node $v \in \mathcal{V}$, in general representing a multidimensional array, and each operation node $o \in \mathcal{O}$, representing a vectorized operation (an operation whose arguments and outputs can be multidimensional arrays as opposed to scalar values). The size of a variable is the number of scalar values in the multidimensional array represented by the variable node. The size of a node $x \in \mathcal{N}$ is zero if $x \in \mathcal{O}$. If $x \in \mathcal{V}$, then its size is given by the product of the dimensions of the n -dimensional array represented by x . Note that operation nodes do not have a notion of size. The analyses that follow use the `sgn` function to indicate whether or not a variable is allocated at some point during program execution. The DAG structure of the graph representation makes it straightforward to compute the execution order of all operations in the model so that all variable values are computed prior to their first use in the program.

To compute the memory requirement for the entire system model, including the implicit operations, recursive functions are defined in the subsections that follow. For the purpose of defining any function that computes memory costs recursively, the

system-level model is contained within an operation $o \in \mathcal{O}$. A function $h : \mathcal{O} \rightarrow \mathcal{N}^n$ provides a mapping from an operation to the list of sorted nodes within the model contained within an operation. The function h is applied to implicit operations and the operation containing the system-level model.

5.2 Analysis of Space Complexity of Model Evaluation at Run Time

The memory cost associated with computing model outputs may be measured by counting the number of scalar values (inputs, outputs, and intermediate variables) allocated at run time. Allocating all scalar values scales poorly as the size of the model (in terms of the number of scalar values in the inputs, outputs, and intermediate variables) increases. To minimize the amount of memory allocated at run time, only the values necessary for continuing to evaluate outputs within the computational model should be allocated, and all values that will no longer be used for the remainder of the execution should be deallocated. The minimum number of scalar values required to compute a given model's outputs can be found by determining the maximum number of scalar values that must be allocated during execution.

In what follows, an analysis of the memory costs at run time is presented. The memory cost in this section only refers to model evaluation, and not evaluation of the derivatives or uncertainties. In reverse mode, all variables must be kept in memory to compute the total derivatives after each simulation. The memory cost of the model variables when computing derivatives in reverse mode is given by

$$\gamma_{\text{rev}}(x, k) = \sum_{i=1}^n \text{size}(x_i). \quad (7)$$

The number of scalar value allocations that program execution requires depends on the size of each variable representing a multidimensional array, as well as the graph structure. In order for an operation to compute the value of a variable, both the arguments to the operation and the outputs that the operation computes must be allocated while the operation is executed. When executing a sequence of operations, the number of scalar values required to be allocated in memory when the k^{th} vectorized operation is computed is given by

$$f(x, k) = \begin{cases} \sum_{i=1}^{k-1} \text{size}(x_i) \text{sgn} \left(\sum_{j=k}^n A_{ij} \right) + \sum_{i=k}^n \text{size}(x_i) A_{ki} & x_k \in \mathcal{O} \\ 0 & x_k \in \mathcal{V} \end{cases}, \quad (8)$$

where $x = (x_1, \dots, x_n)$ is a vector of $n = |\mathcal{N}|$ nodes with $x_i \in \mathcal{N}$, $i \in \{1, \dots, n\}$, $k \in \mathbb{N}$, and A is the adjacency matrix of the graph G after performing a topological sort. The matrix A is a sparse, strictly upper triangular matrix. The order of the elements of x is the order produced by performing a topological sort on the graph G .

The function f computes the amount of memory required for an operation $x_k \in \mathcal{O}$ to execute. Although operations do not have a notion of size, f is nonzero when x_k is an operation, and zero when x_k is a variable. This is due to the fact that only the $x_i \in \mathcal{V}$ within the sums in Equation (8) contribute to the memory required for computing $x_k \in \mathcal{O}$, and not $x_k \in \mathcal{O}$ itself. In the case where $x_k \in \mathcal{O}$, the first term, $\sum_{i=1}^{k-1} \text{size}(x_i) \text{sgn}(\sum_{j=k}^n A_{ij})$ takes into account the memory that must be allocated for all variables on which current and future operations depend must be allocated while operation $x_k \in \mathcal{O}$ is executed. The second term, $\sum_{i=k}^n \text{size}(x_i) A_{ki}$ takes into account the memory that must be allocated for all outputs of the current operation $x_k \in \mathcal{O}$ while the current operation is executed.

In the best case, the amount of memory allocated during program execution reaches a maximum equal to the minimum amount of memory required to complete the program execution. The function f takes into account the memory that must be allocated for the variables connected to operations within a model, but it does not include the memory that must be allocated for the variables within the models contained within the implicit operations $o \in \mathcal{I}$, where $\mathcal{I} \subseteq \mathcal{O}$ is the set of implicit operations contained in the model, described in Section 3.2. Taking into account the additional memory cost incurred by the presence of implicit operations, the total memory cost is given by the recursive function

$$\gamma_{\text{fwd}}(o) = \max\left(\max_{k \in \{1, \dots, n\}} (f(h(o), k)), \max_{(q, k) \in Q} (f(q, k) + \gamma_{\text{fwd}}(q))\right), \quad (9)$$

where $o \in \mathcal{O}$, $Q = \{(q_k, k) : q_k \in \mathcal{N}^n \cap \mathcal{I}\}$ is the set of pairs of implicit operations and their indices in the list of sorted nodes in the model contained within o . If all operations are explicit operations, then the second argument to the outer max function is zero. If any operation is an implicit operation, then it is possible for the memory required to evaluate an implicit operation to dominate the memory allocation requirement, and g must be applied recursively to each implicit operation within the system-level model, as well as any implicit operations contained within the model within an implicit operation.

5.3 Analysis of Space Complexity of Computing Derivatives at Run Time

The computational model generated by the compiler supports fully automating derivative computation for the purpose of applying gradient-based approaches to solving nonlinear programs. Some nonlinear programs may contain coupled, nonlinear systems that do not have closed form solutions. Derivatives are also required in some cases for solving coupled nonlinear systems. Computing derivatives has additional memory requirements beyond those of computing the model outputs. This section presents an analysis of the memory cost of allocating the partial derivatives of all the operations necessary to compute the total derivatives.

The analysis here describes the minimum number of scalar value memory allocations associated with the partial derivatives required for computing total derivatives

using forward- and reverse-mode differentiation. Since the sparsity structure of the partial derivatives for each operation is different across operations, there is not a single equation to compute the minimum number of required scalar value memory allocations. The upper bound on the minimum number of scalar values to allocate in memory to compute the (dense) partial derivatives of the operation represented by the k^{th} node is given by

$$\alpha_{\text{dense}}(x, k) = \sum_{a=1}^{k-1} \sum_{b=k+1}^n \text{size}(x_a) \text{size}(x_b) \text{sgn} \left(\sum_{j=k}^n A_{aj} \right) \text{sgn} \left(\sum_{j=k}^n A_{bj} \right). \quad (10)$$

The right-hand side of (10) computes the number of scalar values that must be allocated to compute the partial derivatives of the operation represented by the k^{th} node assuming that each vector output depends on each vector input. The minimum memory required for allocating the partial derivatives computed while the operation represented by the k^{th} node is $\alpha(x, k) \leq \alpha_{\text{dense}}(x, k)$. This takes into account only the partial derivatives themselves and not the variable values required to be allocated in order to compute the partial derivatives.

For problems such as multidisciplinary design optimization problems where typically the number of design variables is greater than the number of constraints and objective, reverse-mode differentiation is faster than, but more memory-intensive than forward-mode differentiation. The minimum memory cost of computing the partial derivatives using reverse-mode differentiation (not including the model variables used to compute said derivatives) during program execution is given by the recursive function,

$$\beta_{\text{rev}}(o) = \sum_{k=1}^n \alpha(h(o), k) + \sum_{i \in \mathcal{I}} \beta_{\text{rev}}(i). \quad (11)$$

This represents the best-case memory cost of computing total derivatives for the system-level model, assuming reverse-mode differentiation is used for the system-level model and all models nested within implicit operations. While reverse-mode differentiation is faster when the number of model inputs is greater than the number of model outputs, forward-mode differentiation requires allocating less memory regardless of the relative number of inputs and outputs.

The memory required to compute all partial derivatives of all outputs with respect to all inputs for the operation represented by the k^{th} node as well as all partial derivatives that need to be in memory in order to solve for k^{th} block of the total derivatives when using forward-mode differentiation is given by

$$p(x, k) = \begin{cases} \sum_{i=1}^{k-1} \alpha(x, i) \text{sgn} \left(\sum_{j=k+1}^n A_{ij} \right) + \alpha(x, k) & x_k \in \mathcal{O} \\ 0 & x_k \in \mathcal{V} \end{cases}, \quad (12)$$

where $\alpha(x, i)$ computes the memory required for computing the partial derivatives of the i^{th} operation. The function p computes the amount of memory required for evaluating the partial derivatives for a single operation $x_k \in \mathcal{O}$. If x_k is a variable, p is zero. In the case where $x_k \in \mathcal{O}$, the first term, $\sum_{i=1}^{k-1} \alpha(x, i) \text{sgn}(\sum_{j=k+1}^n A_{ij})$ takes into account all partial derivatives on which current and future components of the total derivatives depend must be allocated while operation $x_k \in \mathcal{O}$ is executed. The second term, $\alpha(x, k)$, takes into account the memory required for computing the partial derivatives of the current operation $x_k \in \mathcal{O}$.

The minimum memory cost of computing the partial derivatives required for computing the total derivatives using forward-mode differentiation during program execution is given by the recursive function,

$$\beta_{\text{fwd}}(o) = \max(\max_{k \in \{1, \dots, n\}} (p(h(o), k)), \max_{(q, k) \in Q} (p(q, k) + \beta_{\text{fwd}}(q))). \quad (13)$$

Equations (10), (11), (12), and (13) apply to computing memory costs whether a graph representation represents multidimensional array variables and vectorized operations, or scalar variables and nonvectorized operations. The total memory allocation required for computing the unknown variables (model outputs) is presented in Section 5.2.

The total memory allocation required for computing both the unknown variables (model outputs) and the partial derivatives necessary to compute the total derivatives, is given by the recursive function

$$\delta_{\text{fwd}}(o) = \max(\max_{k \in \{1, \dots, n\}} (f(h(o), k) + p(h(o), k)), \max_{(q, k) \in Q} (f(h(q), k) + p(q, k) + \delta_{\text{fwd}}(q))). \quad (14)$$

Equations (7), (8), (9), and (14) apply to computing memory costs whether a graph representation represents multidimensional array variables and vectorized operations, or scalar variables and nonvectorized operations.

5.4 Analysis of Space Complexity at Compile Time

At compile time, the compiler generates a graph representation of the model where each node represents a multidimensional array variable or a vectorized operation. The graph representation has a lower compile time memory cost relative to that of a graph representation whose nodes each represent a scalar value or operation that operates on scalar values. This section presents the compile time benefits to using the vectorized graph representation of Section 3.2.

The relationship between the size of the graph representation used in the compiler and the graph representation whose nodes represent scalar values and operations is highly dependent on the model. The number of scalar values represented per variable node in the graph representation in Section 3.2 is at least

$$\sum_i^n \text{size}(v_i), \quad (15)$$

where n is the number of nodes in the graph representation and $v_i \in \mathcal{V}$ are variable nodes representing multidimensional arrays. There are, however, other intermediate values in, for example, linear algebra expressions that are not represented by more nodes in the graph representation.

A computational graph that represents each scalar value with a node may be recovered from a graph representation, but the exact mapping from a graph representation to a scalar representation depends on each operation. In the simplest cases, the number of nodes in a computational graph per operation is given by

$$\sum_i \sum_j \text{size}\left(\frac{\partial b_i}{\partial a_j}\right), \quad (16)$$

where a_i, b_j are the (multidimensional array) inputs and outputs, respectively of a given (vectorized) operation. In some cases, $\frac{\partial b_i}{\partial a_j} = 0$ for a given input-output pair, leading (16) to be an overestimate of the number of nodes in the computational graph. In other cases, (16) is an underestimate. For example, linear algebra operations such as $y = Ax + b$, where a node representing each intermediate scalar value for each scalar multiplication and addition of each row of A with x is also part of the graph representation. In the graph representation, there is one node for each, b, A, x, Ax , and y , regardless of the dimensions of each variable. If each node in the computational graph is a neuron in the computational graph for an artificial neural network (ANN), then more than one node in the graph representation will correspond to a node/neuron in the computational graph, since each neuron is a sum of inputs followed by an activation function, which would require three nodes in the graph representation, one for the sum operation, one for the result of the sum, and one for the activation function.

The smaller graph representation does not only reduce memory cost at compile time. Unlike most expression graphs used in AD libraries, the graph representation does not need to be kept in memory at run time.

5.5 Analysis of Time Complexity at Run Time

The minimum necessary time to evaluate a computational model can be achieved by taking advantage of all opportunities for parallelization. When taking advantage of all opportunities for parallelization, the run time depends on the length of the longest path, or *critical path* of the directed acyclic graph (DAG). Directed paths (Definition 2) with common start and end points, and no other nodes in common may be traversed in parallel. If multiple paths are traversed in parallel, then the time required to execute all operations is determined by the number and execution time of operations on the critical path, since operations that are on different paths may be executed in parallel.

This section presents an analysis of the time cost of evaluating a computational model when taking advantage of all opportunities for parallelization.

To compute the execution time of a fully parallelized implementation, let $G'(\mathcal{N}', \mathcal{E}')$ be a DAG, where the edge set $\mathcal{E}' = \{(u, w) : (u, v), (v, w) \in \mathcal{E}, v \in \mathcal{O}\}$ represents dependencies between variables ($u, w \in \mathcal{V}$ for each $(u, w) \in \mathcal{E}'$) and $\mathcal{N}' = \{v : v \in \mathcal{V}\}$ represents variables. Let $t : \mathcal{E}' \rightarrow \mathbb{R}_+$ map each edge to a value representing the computation time of an operation $v \in \mathcal{O}$. Then the critical path of the weighted graph (G', t) , serves as an estimate of the execution time of the simulation when exploiting all opportunities for parallelization. If $t = 1$, then the time to compute all operations is simply the number of operations on the critical path.

Consider the following. If an edge $e \in \mathcal{E}'$ corresponds to an operation $v \in \mathcal{J}$, where $\mathcal{J} \subseteq \mathcal{O}$ is the set of explicit operations, then the time t for the operation represented by either e or v to execute is $t(e) = 1$. If an edge $e \in \mathcal{E}'$ corresponds to an operation $v \in \mathcal{I}$, where $\mathcal{I} \subseteq \mathcal{O}$ is the set of implicit operations, then the time t for the operation represented by $e \in \mathcal{E}'$ to execute is as high as $t(e) = T(h(v))m_v$, where h is described at the end of Section 5, $m_v \in \mathbb{N}$ is the maximum number of iterations used to solve the implicit function represented by v , and

$$T(o) = \sum_{i=1}^k t(x_i) \quad (17)$$

is the total time to evaluate all operations in series within a model contained within the operation $x \in \mathcal{O}$. To compute an estimate of the execution time exploiting all available opportunities for parallelization, only the $x_i \in \mathcal{O}$ on the critical path need to be included in Equation (17).

6 The Computational System Design Language

This section presents an overview of a new algebraic modeling language (AML) called the Computational System Design Language (CSDL). CSDL is a compiled, embedded domain-specific language (EDSL), which inherits the syntax of its host language, Python, a popular language used for scientific computing. CSDL provides an interface for the input to the front end in the novel methodology presented in Section 3. CSDL is designed to support natural mathematical expressions and hide implementation details for derivative computation.

The three defining features of CSDL are (1) an object-oriented syntax for defining a model hierarchy, (2) a functional syntax for defining relationships between variables, and (3) an implicit operation type containing a model that defines residuals that must be converged when solving coupled systems of equations. All CSDL users interact with three main Python classes that comprise the CSDL compiler: `Model`, `GraphRepresentation`, and `Simulator`. Users can optionally use the `CustomExplicitOperation` and `CustomImplicitOperation` to define functions that are not available in the CSDL standard library.

Listing 1 shows the compilation process in CSDL. Detailed documentation for the Computational System Design Language (CSDL) is available at the CSDL website, <https://lsdolab.github.io/csdl/>. Links to the GitHub repository containing the compiler implementation, including the CSDL compiler front end and compiler middle end (<https://github.com/lsdolab/csdl>) along with a list of links to available back ends are also provided within the CSDL website.

Listing 1 The compilation process using CSDL

```
# Import language and compiler front end
from csdl import Model, GraphRepresentation
# Import compiler back end
from back_end_package import Simulator

# Define numerical model
class Example(Model):
    def define(self):
        # ...

# Generate graph representation
rep = GraphRepresentation(Example())
# Generate computational model
sim = Simulator(rep)
# Evaluate computational model
sim.run()
# Print results
print(sim[" ... "])
```

Model

The first class users interact with is the `Model` class, which users extend by defining subclasses that define the numerical model and the optimization problem. Within each `Model` subclass, users define relationships between variables. Relationships between variables may be defined as connections between variables in different instances of user-defined `Model` subclasses contained within a user-defined `Model` subclass (as shown in Listing 2), as well as functions or compositions of functions (as shown in Listing 3). The CSDL standard library provides standard functions that are continuous and differentiable in their arguments (e.g. trigonometric functions, exponential function). Users do not have to implement derivatives for any function in the standard library, facilitating the use of a `Model` subclass definition for use in an optimization workflow.

Listing 2 Hierarchical composition in CSDL

```
class Example(Model):
    def define(self):
        self.add(Aerodynamics(), name='aerodynamics')
        self.add(Structures(), name='structures')
```

```

# form connections between variables in different models
self.connect('aerodynamics.lift', 'structures.force_z')
self.connect('aerodynamics.drag', 'structures.force_y')

```

Listing 3 Functional composition in CSDL

```

class Example(Model):
    def define(self):
        p = self.declare_variable('p', shape=(2,))
        x = p[0]
        y = p[1]
        distance = (x**2 + y**2)**(1/2)
        self.register_output('distance', distance)

```

GraphRepresentation

The **GraphRepresentation** class contains information for the directed acyclic graph for the model that is central to the novel methodology presented in Section 3.2. The **GraphRepresentation** class constructor serves as the compiler front end, which constructs the directed acyclic graph from an instance of a user-defined **Model** subclass. Both the **Model** class and the **GraphRepresentation** class are defined in a Python package called **csdl**. The **csdl** package also provides functions that users can apply to a **GraphRepresentation** object to perform implementation-independent optimizations (Section 4), forming the compiler middle end.

Simulator

The **Simulator** class is a container for the computational model generated from a **GraphRepresentation** object. The CSDL compiler generates executable code from an instance of the **GraphRepresentation** class by constructing an instance of the **Simulator** class. A package separate from the **csdl** package supplies the **Simulator** class, whose constructor serves as the compiler back end, which generates the executable code that evaluates the computational model. Users can select a back end from any of the available back ends to generate a computational model based on advantages that one back end might have over another, without modifying the model definition.

A compiler back end is free to compile directly to low-level machine code using whatever strategy the compiler back end developer(s) may use. A compiler back end is also free to generate code in any high performance general-purpose language (e.g. C, C++, Fortran)⁵, and rely on that language’s compiler to generate executable code. The **Simulator** class interface connects to an external solver to solve the optimization problem defined in the numerical model and also provides access to data at run time, so users can also print, plot, and save run time values after a simulation terminates.

⁵Back ends that generate code in these languages do not currently exist, but can be developed to use CSDL **GraphRepresentation** objects generated by the CSDL compiler front end.

CustomExplicitOperation and CustomImplicitOperation

In some cases, users may need to define their own custom relationships between variables and operate at a lower level of abstraction by implementing imperative/procedural code to compute output values as well as the partial derivatives of the function. Custom implementations also have the potential to provide performance improvements over automatically generated gradients. CSDL provides the `CustomExplicitOperation`, and `CustomImplicitOperation` classes for defining custom relationships not defined in the standard library. The

`CustomExplicitOperation` is the base class provided for users to define their own custom explicit functions and their partial derivatives when a function is not available in the standard library. Each `CustomExplicitOperation` subclass can also provide an interface to external analysis tools to integrate into a larger multidisciplinary optimization. `CustomImplicitOperation` is similar to `CustomExplicitOperation` except that it is used for defining implicit functions whose residuals cannot be defined using a composition of functions from the standard library.

7 Results

This section presents results from applying the new methodology presented in Section 3 to practical models used in engineering design. Section 7.1 presents results comparing the number of lines of code between OpenMDAO (Section 2.3) and CSDL of Section 6. Section 7.2 presents properties of nine models based on applying the analyses of Section 5 to the graph representations of the nine models.

7.1 Quantifying the Impact of Automating Sensitivity Analysis

This section compares the number of lines of code provided by the user to generate a computational model between two packages written in CSDL and two similar packages written in OpenMDAO: `OpenAeroStruct` [36, 37], and `lsdo_bemt` [38]. The difference in the number of lines of code quantifies the impact of automating sensitivity analysis. The `OpenAeroStruct` package is a tool for performing aerostructural optimization, which uses a vortex-lattice method (VLM) and a beam finite-element method. The `lsdo_bemt` package, written in OpenMDAO, implements a blade element momentum (BEM) method for designing propellers and turbines. Figure 7 shows the difference in the number of lines of code used to define the models in each project.

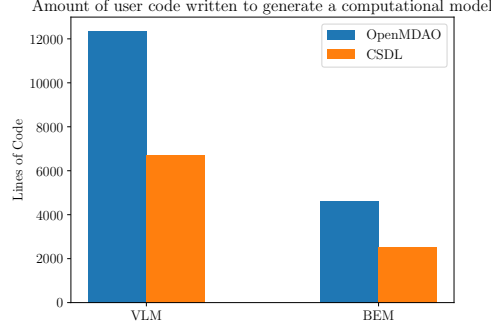


Fig. 7 The number of lines of code required to implement or generate a computational model using OpenMDAO is $\sim 2\times$ greater than that of using CSDL. OpenMDAO is a software framework that uses the current state-of-the-art method for exact scalable sensitivity analysis of complex models.

The CSDL models use approximately half the number of lines of code to define the same model as in OpenMDAO. The reason for this is due to the fact that CSDL automates both partial and total derivative calculations, whereas OpenMDAO requires that users provide implementations for partial derivative calculations. In this sense, OpenAeroStruct and `lsdo_bemt` also include implementation details that CSDL code does not, mixing different levels of abstraction, and reducing the overall readability and maintainability of the model code.

7.2 Run-Time Costs

This section presents the results of estimating the run time costs of various system models based on the analyses presented in Section 5. The time and memory costs for nine engineering models are estimated based on their graph representations of Section 3.2. Figures 8, 9, 11, and 12 are based on nine engineering models defined using the Computational System Design Language (CSDL) described in Section 6: a model of a propeller rotor using blade element momentum (BEM) theory [39], a model of a propeller rotor using the blade element momentum-based ideal-loading design (BILD) method [40], a model of airflow over an electrical vertical takeoff and landing (eVTOL) vehicle concept using the vortex lattice method (VLM) [41], an equivalent circuit model (ECM) of a battery [41], a model of a propeller rotor using a Pitt-Peters approach, a model of a laser-powered unmanned aerial vehicle (UAV) concept, a model of a permanent magnet synchronous motor (PMSM) [42], a model used for general aviation aircraft weight estimation (“Weights”) [41], and a model of wing, rotor, fuselage, and tail geometries with mesh generation (“Geometry”) [41].

Figure 8 shows the relative time cost of evaluating a computational model when exploiting all opportunities for parallelization⁶ for each of the nine models presented in this section. The compiler can compute these estimates of the parallelized execution time by applying Equation (17) to the graph representation of a model. Compiler back

⁶Execution of individual vectorized operations may be further parallelized, reducing computation time by more than what is shown in Figure 8.

end developers can use the graph representation to generate a target for the efficiency of generated parallelized computational models, and users can take advantage of this information to choose whether or not to exploit opportunities for parallelization if the compiler back end of their choice offers that capability.

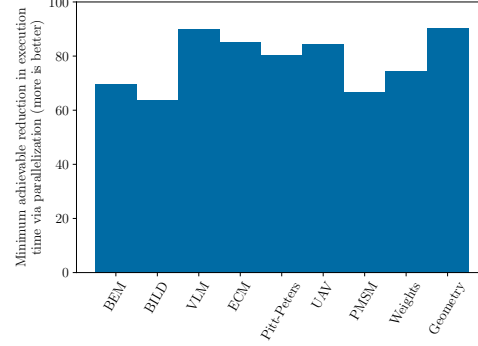


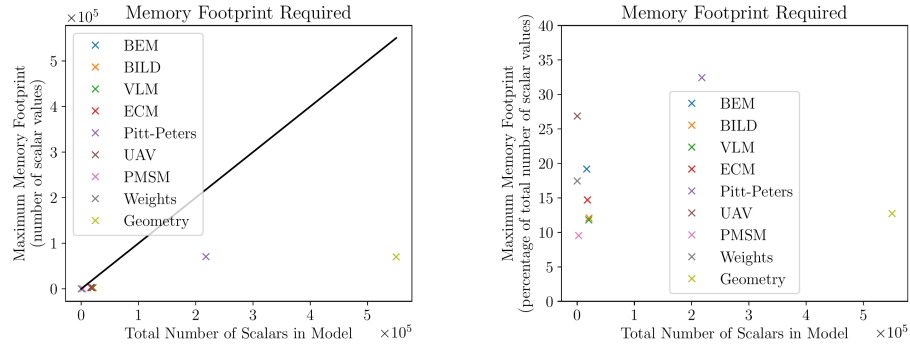
Fig. 8 The graph representation for each model is used to predict the time to evaluate a computational model using all opportunities for parallelization as a percentage of the time to evaluate a computational model without exploiting any opportunities for parallelization. Note that all operations are assumed to take the same amount of time to execute.

Figures 9, 11, and 12 show an estimate of the memory footprint required for evaluating the generated computational model as measured by the number of scalar values. Only the memory required to evaluate a computational model is computed, and the memory required for derivative computation is not included. Figure 9 shows the total number of scalar values used to define each model versus an estimate of the required memory footprint as measured by the maximum number of scalar values and the maximum percentage of the total number of scalar values used to define each model. Each point in Figures 9 and 11 is computed by applying Equation (9) to the graph representation for each of the nine engineering models under consideration.

Figure 9 shows how (a) the maximum number of scalar values does not increase rapidly and levels off as the total number of scalar values used to define the model increases and (b) this number stays below 35% of the total number of scalar values used to define the model. These trends suggest that the memory required for evaluating computational models scales well with the total number of scalar values used to define each model.

Figure 11 shows the adjacency matrix density for the graph representation of each system-level model versus an estimate of the percentage of the total number of scalar values to allocate. The density of the adjacency matrices for all nine engineering models is less than 1%. The percentage of total scalar values to allocate shows a positive correlation (albeit small) with the adjacency matrix density. This is expected since a larger number of arguments per operation increases the required memory footprint.

Figure 10 shows visualizations and results from three low-fidelity models using a compiler front end that generates a graph representation from code written in the



(a) Maximum number of scalar values to allocate at any point during program execution versus the total number of scalar values used to define the model; the straight line shows the number of scalar values allocated if the total number of scalar values used to define the model are allocated.

(b) Maximum percentage of the total number of scalar values used to define the model to allocate at any point during program execution versus the total number of scalar values used to define the model

Fig. 9 Maximum number of scalar values to allocate at any point during program execution.

Computational System Design Language (CSDL) and a compiler back end that generates OpenMDAO code. Figure 10 also shows the adjacency matrices of the graph representation for each system-level model.

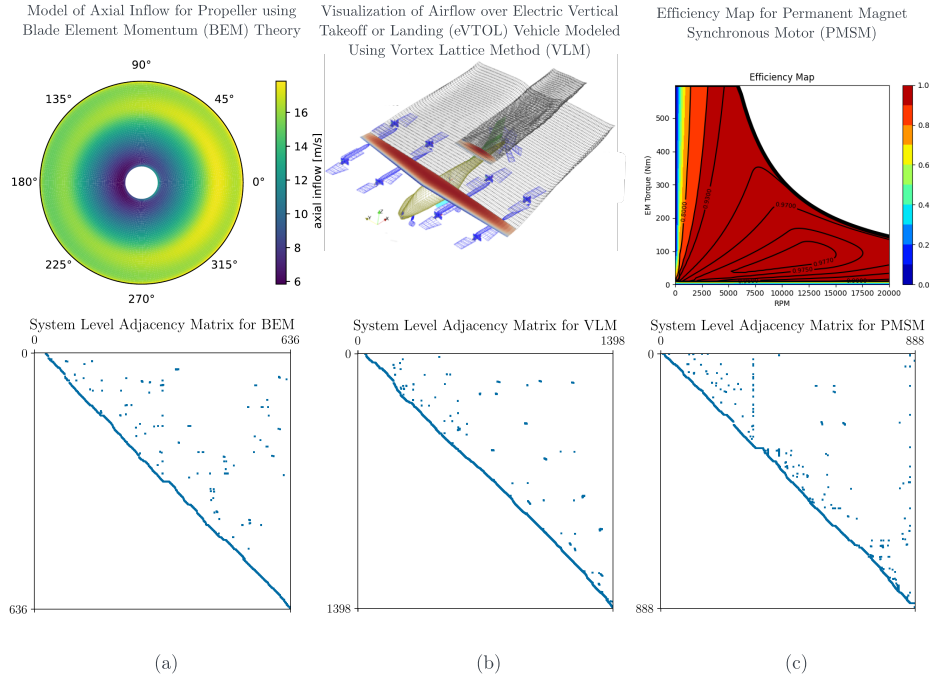


Fig. 10 Three models generated using CSDL and the system-level adjacency matrices corresponding to their compiler graph representations: (a) axial inflow velocity for a propeller modeled using blade element momentum (BEM) theory [40], (b) airflow over an electrical vertical takeoff and landing (eVTOL) vehicle concept using the vortex lattice method (VLM) [41], and (c) an efficiency map for a permanent magnet synchronous motor (PMSM) [42]. The computational models corresponding to the images in the top row were compiled using a compiler back end that constructs an OpenMDAO model.

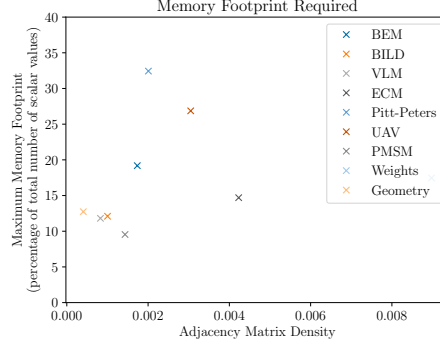


Fig. 11 Percentage of the total number of scalar values used to define the model that are allocated during program execution versus the density of the adjacency matrix of the model's graph representation. The density remains below 1% for all models examined, and there is a small positive correlation between overall memory footprint and the density of the adjacency matrix.

Figure 12 shows an estimate of the maximum number of scalar values to allocate at any point during model evaluation as a function of the percentage of the operations executed in a computational model during run time. The final memory cost depends on the number of outputs the user chooses to keep in memory for analysis or as objective and constraint values after the model evaluation is complete. The model outputs account for the rise in memory requirement until model evaluation is complete. The sharp spikes and drops in Figure 12 indicate opportunities for dynamically allocating chunks of memory temporarily during model evaluation.

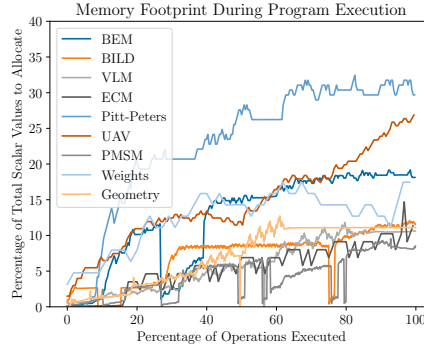


Fig. 12 Number of scalar values that must be allocated at each vectorized operation during model evaluation. The number of values that must be allocated rises to the number of values used to define model outputs (e.g. objective and constraints). These values must remain in memory until the model is evaluated to provide external access to the resulting values.

In addition to computing an estimate of the memory cost of evaluating a computational model without computing derivatives, the compiler can also compute an estimate of the memory cost when evaluating the computational model when using

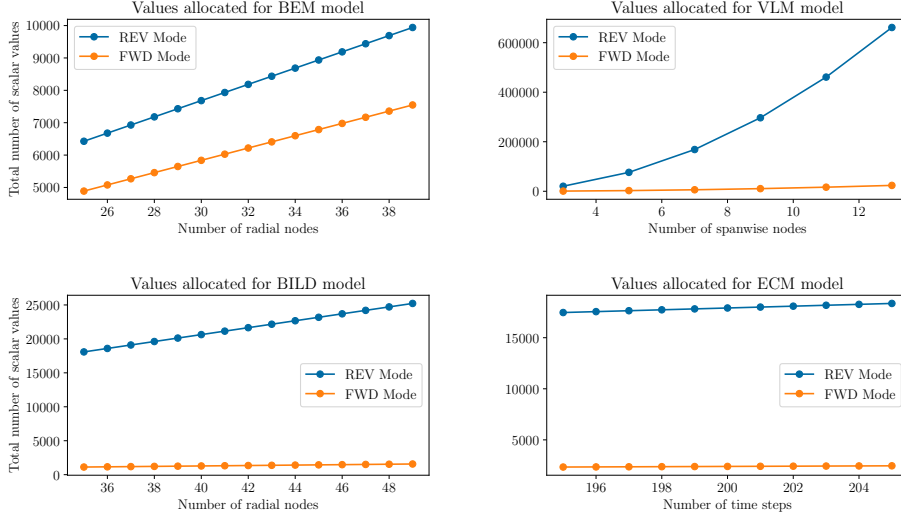


Fig. 13 Required memory at run time computed by applying Equations (7) and (9) to the graph representation (Section 3.2) for various models defined in CSDL, as a function of a key parameter; clockwise from top left: a model of a propeller rotor using blade element momentum (BEM) theory; an aerodynamic model of an electrical vertical takeoff and landing (eVTOL) vehicle concept using the vortex lattice method (VLM); an equivalent circuit model (ECM) of a battery; and a model of a propeller rotor using BEM-based ideal-loading design (BILD) method.

forward or reverse-mode differentiation. Figure 13 shows a comparison of memory cost estimates for existing models written in CSDL, depending on whether they use forward-mode differentiation or reverse-mode differentiation.

The difference in memory requirements between forward mode and reverse mode highlights a trade-off between speed and memory efficiency for both methods. The derivative computation time is proportional to the number of inputs in the case of forward-mode differentiation, and proportional to the number of outputs in the case of reverse-mode differentiation. If the number of outputs (e.g. objective and constraints) is greater than the number of inputs (e.g. design variables), then there is no trade-off, and forward-mode differentiation is the obvious choice. If the number of inputs is greater than the number of outputs, then computation time for computing total derivatives scales better using reverse-mode differentiation, but the memory requirement is greater.

These memory requirements may guide the user to select different model parameters to generate a simulation of a model with different levels of fidelity to accommodate available resources, depending on the problem. For example, Figure 13 shows how increasing the number of spanwise nodes increases the memory requirement quickly when solving an optimization problem containing a vortex lattice method (VLM) model using reverse-mode differentiation, but not forward mode. In this example, increasing the fidelity of the model may require using forward-mode differentiation if resources are not available to accommodate the increased memory requirement

imposed by using reverse-mode differentiation. On the other hand, increasing the number of time steps in the ECM model in Figure 13 does not have such a dramatic effect on the memory requirement. In this example, increasing fidelity does not dramatically increase the demand for resources, so reverse-mode differentiation may remain a viable option, even if the number of time steps is very large.

8 Conclusion and Future Work

Efficient derivative computation for tightly coupled, multidisciplinary systems presents a challenge for implementing computational models for use within a gradient-based optimization framework. The adjoint method is a powerful technique for computing derivatives in systems with a large number of design variables. Various languages, frameworks, and libraries have attempted to automate derivative computation, achieving some combination of: partially automated, exact derivative computation; fully automated, approximate derivative computation; and high performance. Existing approaches do not achieve both full automation and high performance for multidisciplinary design optimization problems.

This paper presented a novel methodology for constructing computational models that fully automates adjoint-based sensitivity analysis. The new methodology centers around constructing a graph representation of a numerical model, applying a series of transformations to the graph representation, and generating a computational model that automatically computes derivatives. This methodology was implemented as a three-stage compiler.

The implementation of the three-stage compiler motivated the design of a new algebraic modeling language (AML) called the Computational System Design Language (CSDL). CSDL provides an interface to input a numerical model to the compiler front end in the novel methodology. For two engineering models, the use of CSDL showed a reduction in the number of lines of code required to implement a computational model by $\sim 2\times$.

Best-case complexity analyses required for model evaluation when applying the new methodology were presented as well. The complexity analyses were applied to the graph representations of nine practical models used in engineering design. Estimates of the memory required for evaluating the computational model scale well with the total number of scalar values used in defining the numerical model, but more data is required to build an empirical model of the relationship between the number of scalar values used to define the numerical model and the total memory required for evaluating the computational model. Estimates of the time cost of evaluating the computational model when exploiting all opportunities for parallelization show significant savings in computation time compared to the time cost of evaluating the operations within the computational model in series.

The numerical model's graph representation itself offers new opportunities for gaining insight into more efficient implementations of computational models. Plans for future work consist of (1) building on current results using a wider array of models to develop predictive models of computational efficiency based on model structure, (2) using the graph representation to identify the type of the optimization problem and

recommend a solver to the user based on the optimization problem type, and (3) developing a new compiler back end that generates computational models whose efficiency in time and memory approaches the estimates presented in this paper.

9 Acknowledgments

The authors also thank Marius Ruh, Jiayao Yan, and Luca Scotzionovsky for helping generate the data in

10 Conflict of Interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

11 Funding

This work was supported in part by the National Science Foundation under grant no. 1936557 and NASA grant number 80NSSC21M0070.

12 Ethical Approval

Not Applicable

13 Informed Consent

Not Applicable

14 Replication of Results

Detailed documentation for the Computational System Design Language (CSDL) is available at the CSDL website, <https://lsdolab.github.io/csdl/>. Links to the GitHub repository containing the compiler implementation, including the CSDL compiler front end and compiler middle end (<https://github.com/lsdolab/csdl>) along with a list of links to available back ends are also provided within the CSDL website.

References

- [1] Mernik, M., Heering, J. & Sloane, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* **37**, 316–344 (2005).
- [2] Kallrath, J. *Modeling Languages in Mathematical Optimization* (Springer Science & Business Media, 2004).
- [3] Kallrath, J. *Algebraic Modeling Systems: Modeling and Solving Real World Optimization Problems* Vol. 104 (Springer Science & Business Media, 2012).

- [4] Hart, W. E., Watson, J.-P. & Woodruff, D. L. Pyomo: modeling and solving mathematical programs in python. *Mathematical Programming Computation* **3**, 219–260 (2011).
- [5] Bynum, M. L. *et al. Pyomo—optimization modeling in python* Third edn, Vol. 67 (Springer Science & Business Media, 2021).
- [6] Dunning, I., Huchette, J. & Lubin, M. Jump: A modeling language for mathematical optimization. *SIAM review* **59**, 295–320 (2017).
- [7] Fourer, R., Gay, D. M. & Kernighan, B. W. A modeling language for mathematical programming. *Management Science* **36**, 519–554 (1990).
- [8] Bussieck, M. R. & Meeraus, A. in *General algebraic modeling system (gams)* 137–157 (Springer, 2004).
- [9] Bisschop, J. & Meeraus, A. in *On the development of a general algebraic modeling system in a strategic planning environment* 1–29 (Springer, 1982).
- [10] Åkesson, J. *Optimica—an extension of modelica supporting dynamic optimization* (2008).
- [11] Fishwick, P. A. A taxonomy for simulation modeling based on programming language principles. *IIE transactions* **30**, 811–820 (1998).
- [12] Sinha, R., Paredis, C. J., Liang, V.-C. & Khosla, P. K. Modeling and simulation methods for design of engineering systems. *J. Comput. Inf. Sci. Eng.* **1**, 84–91 (2001).
- [13] Viklund, L. & Fritzson, P. Objectmath—an object-oriented language and environment for symbolic and numerical processing in scientific computing. *Scientific Programming* **4**, 229–250 (1995).
- [14] Fritzson, P. & Engelson, V. *Modelica—a unified object-oriented language for system modeling and simulation*, 67–90 (Springer, 1998).
- [15] Gray, J., Moore, K. & Naylor, B. *Openmdao: An open source framework for multidisciplinary analysis and optimization*, 9101 (2010).
- [16] Gray, J. S., Hwang, J. T., Martins, J. R., Moore, K. T. & Naylor, B. A. Openmdao: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization* **59**, 1075–1104 (2019).
- [17] Alnæs, M. S., Logg, A., Ølgaard, K. B., Rognes, M. E. & Wells, G. N. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)* **40**, 1–37 (2014).

- [18] Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM review* **59**, 65–98 (2017).
- [19] Revels, J., Lubin, M. & Papamarkou, T. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892* (2016).
- [20] Baydin, A. G., Pearlmutter, B. A., Radul, A. A. & Siskind, J. M. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* **18**, 1–43 (2018).
- [21] Kirby, R. C. & Logg, A. A compiler for variational forms. *ACM Transactions on Mathematical Software (TOMS)* **32**, 417–444 (2006).
- [22] Rognes, M. E., Kirby, R. C. & Logg, A. Efficient assembly of $h(\text{div})$ and $h(\text{curl})$ conforming finite elements. *SIAM Journal on Scientific Computing* **31**, 4130–4151 (2010).
- [23] Ølgaard, K. B. & Wells, G. N. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software (TOMS)* **37**, 1–23 (2010).
- [24] Logg, A., Ølgaard, K. B., Rognes, M. E. & Wells, G. N. in *Ffc: the fenics form compiler* 227–238 (Springer, 2012).
- [25] Alnæs, M. S. & Mardal, K.-A. On the efficiency of symbolic computations combined with code generation for finite element methods. *ACM Transactions on Mathematical Software (TOMS)* **37**, 1–26 (2010).
- [26] Alnæs, M. S. & Mardal, K.-A. in *Syfi and sfc: Symbolic finite elements and form compilation* 273–282 (Springer, 2012).
- [27] Hwang, J. T. & Martins, J. R. A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives. *ACM Transactions on Mathematical Software (TOMS)* **44**, 1–39 (2018).
- [28] Martins, J. R. & Hwang, J. T. Review and unification of methods for computing derivatives of multidisciplinary computational models. *AIAA journal* **51**, 2582–2599 (2013).
- [29] Lattner, C. & Adve, V. *Llvm: A compilation framework for lifelong program analysis & transformation*, 75–86 (IEEE, 2004).
- [30] Margossian, C. C. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery* **9**, e1305 (2019).
- [31] Moses, W. & Churavy, V. Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F. & Lin, H. (eds) *Instead of rewriting foreign code for machine learning*,

automatically synthesize fast gradients. (eds Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F. & Lin, H.) *Advances in Neural Information Processing Systems*, Vol. 33, 12472–12485 (Curran Associates, Inc., 2020).

- [32] Moses, W. S. *et al.* *Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme*, SC '21 (Association for Computing Machinery, New York, NY, USA, 2021).
- [33] Moses, W. S. *et al.* *Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation*, SC '22 (IEEE Press, 2022).
- [34] Kahn, A. B. Topological sorting of large networks. *Communications of the ACM* **5**, 558–562 (1962).
- [35] Squire, W. & Trapp, G. Using complex variables to estimate derivatives of real functions. *SIAM review* **40**, 110–112 (1998).
- [36] Jasa, J. P., Hwang, J. T. & Martins, J. R. R. A. Open-source coupled aerostuctural optimization using Python. *Structural and Multidisciplinary Optimization* **57**, 1815–1827 (2018).
- [37] Chauhan, S. S. & Martins, J. R. R. A. *Low-fidelity aerostuctural optimization of aircraft wings with a simplified wingbox model using OpenAeroStruct*, 418–431 (Springer, Lisbon, Portugal, 2018).
- [38] Ha, T. H., Lee, K. & Hwang, J. T. *Large-scale multidisciplinary optimization under uncertainty for electric vertical takeoff and landing aircraft* (2020).
- [39] Hwang, J. T. & Ning, A. *Large-scale multidisciplinary optimization of an electric aircraft for on-demand mobility*, 1384 (2018).
- [40] Ruh, M. L. & Hwang, J. T. *Robust modeling and optimal design of rotors using blade element momentum theory*, 2598 (2021).
- [41] Sarojini, D. *et al.* *Large-scale multidisciplinary design optimization of an evtol aircraft using comprehensive analysis* (2023).
- [42] Cheng, Z. *et al.* *A differentiable method for low-fidelity analysis of permanent-magnet synchronous motors* (2023).