UNIVERSITY OF CALIFORNIA SAN DIEGO

Enhancing Multidisciplinary Design Optimization through Automated Computational Model Construction and Sensitivity Analysis

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Engineering Sciences (Aerospace Engineering)

by

Victor Gandarillas

Committee in charge:

Professor John T. Hwang, Chair
Professor Thomas R. Bewley
Professor Ranjit Jhala
Professor Aaron J. Rosengren

2023

The Dissertation of Victor Gandarillas is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

For Emily,

my parents,

my extended family,

and my friends.

EPIGRAPH

Far better it is to dare mighty things, to win glorious triumphs, even though checkered by failure, than to rank with those poor spirits who neither enjoy nor suffer much, because they live in the gray twilight that knows not victory nor defeat.

—Theodore Roosevelt

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Many people contributed to making this dissertation possible. First, I would like to thank my best friend and partner, Emily Winokur. You have been there for me every day of my Ph.D. work. Every day with you has been a gift, and I'm so glad we get to make our biggest achievements so far in life together. I can't wait to see what we do next.

Second, I would like to thank my parents for encouraging me to pursue my dreams and supporting me throughout the ups and downs in my life. I could not have made it where I am today without their love and support.

Third, I am extremely grateful to all of my previous mentors at NASA Armstrong Flight Research Center and the Jet Propulsion Laboratory for their part in teaching me valuable lessons that prepared me for my career in graduate school.

Fourth, I would like to thank my friend Sean Kross for giving his perspective as a computer scientist on my approach to software development and programming language design.

I would also like to thank my lab mates and co-authors Anugrah Jo Joshy, Mark Z. Sperry, and Alex K. Ivanov for their support in developing the Computational System Design Language (CSDL) and its compiler, both of which form a large part of this dissertation. Anugrah and Alex built the standard library for CSDL, and Mark served as a second set of eyes to help complete one of the last features of the CSDL compiler front end.

I would like to thank my committee for their time and advice, especially Thomas R. Bewley, who introduced me to my advisor John T. Hwang. I'd like to thank Ranjit Jhala and Aaron J. Rosengren for their insight and advice on my research topics. Lastly, I'd like to thank my committee chair and my advisor John T. Hwang, for giving me guidance, advice, and ideas, as well as always pushing me to meet a higher standard so I could accomplish this difficult goal.

This dissertation would not have been possible without the collective efforts, guidance, and encouragement of all those mentioned above. To all my friends and coworkers over the years, including those not mentioned here, please accept my sincere thanks for the roles you've played in shaping this academic achievement.

Chapter 2, in part, has been submitted for publication of the material as it may appear in Structural and Multidisciplinary Optimization 2023. Gandarillas, Victor; Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT, Springer Nature, 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, has been submitted for publication of the material as it may appear in Structural and Multidisciplinary Optimization 2023. Gandarillas, Victor; Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT, Springer Nature, 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, has been submitted for publication of the material as it may appear in Structural and Multidisciplinary Optimization 2023. Gandarillas, Victor; Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT, Springer Nature, 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is currently being prepared for submission of the material. Gandarillas V, Hwang JT; American Institute of Aeronautics and Astronautics, 2023. The dissertation author was the primary investigator and author of this paper.

VITA

2013	B. S. in Aeronautics and Astronautics, Purdue University
2018	M. S. in Engineering Sciences (Aerospace Engineering), University of California San Diego
2023	Ph. D. in Engineering Sciences (Aerospace Engineering), University of California San Diego

PUBLICATIONS

Victor Gandarillas and John T. Hwang, "Large-scale multidisciplinary design optimization of a virtual-telescope CubeSat swarm," *Journal of Spacecraft and Rockets*, (in preparation).

Victor Gandarillas, Anugrah Jo Joshy, Mark Z. Sperry, Alexander K. Ivanov, and John T. Hwang, "A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis," *Structural and Multidisciplinary Optimization*, (in revisions).

Victor Gandarillas and John T. Hwang, "Large-scale multidisciplinary design optimization of a virtual-telescope CubeSat swarm," *AIAA Aviation 2023 Forum*, 4021, 2023.

Joshy, Anugrah Jo, Dunn, Ryan, Sperry, Mark, Gandarillas, Victor E, and Hwang, John T., "An SQP algorithm based on a hybrid architecture for accelerating optimization of large-scale systems," *AIAA Aviation 2023 Forum*, 4263, 2023.

Sarojini, Darshan and Ruh, Marius L and Joshy, Anugrah Jo and Yan, Jiayao and Ivanov, Alexander K and Scotzniovsky, Luca and Fletcher, Andrew H and Orndorff, Nicholas C and Sperry, Mark and Gandarillas, Victor E and others, "Large-Scale Multidisciplinary Design Optimization of an eVTOL Aircraft using Comprehensive Analysis," *AIAA Scitech 2023 Forum*, 146, 2023.

Wang, Bingran and Sperry, Mark and Gandarillas, Victor E and Hwang, John T., "Efficient uncertainty propagation through computational graph modification and automatic code generation," *AIAA Aviation 2022 Forum*, 3997, 2022.

Victor Gandarillas and John T Hwang, "Automatic generation of numerical methods for time

integration in large-scale MDO," *AIAA Aviation 2019 Forum*, 3664, 2019.

Gandarillas V. "Singular perturbation method for multiagent coverage control using time-varying density functions." University of California San Diego; 2018.

ABSTRACT OF THE DISSERTATION

Enhancing Multidisciplinary Design Optimization through Automated Computational Model Construction and Sensitivity Analysis

by

Victor Gandarillas

Doctor of Philosophy in Engineering Sciences (Aerospace Engineering)

University of California San Diego, 2023

Professor John T. Hwang, Chair

Multidisciplinary design optimization (MDO) is an approach that uses optimization methods to design complex engineering systems involving multiple disciplines simultaneously. The coupled nature of multidisciplinary systems and the large number of design variables involved in complex systems present unique challenges to solving MDO problems. One of these challenges is the implementation of software necessary to evaluate multidisciplinary models within an optimization framework. When gradient-based optimization approaches are used, efficient and accurate derivatives must be computed for each model evaluation. This dissertation presents an approach that significantly reduces the manual effort required to implement computational

models for use within a gradient-based MDO framework, especially in large-scale problems. This dissertation introduces a novel approach to address these challenges and automate the process, enabling accurate and efficient adjoint-based sensitivity analysis for gradient-based MDO in particular. To address these challenges, a three-stage compiler methodology is proposed. The methodology centers around a graph representation that provides a foundation for automating sensitivity analysis in MDO. In addition, the Computational System Design Language (CSDL) is introduced, which allows for a concise description of the physical system. The adoption of CSDL demonstrates a twofold reduction in code complexity for engineering models, significantly reducing the barrier to entry for MDO practitioners. The three-stage compiler also provides users of CSDL with the ability to measure the effect of the model structure on run-time performance and memory complexity using the graph representation. Finally, the methodology developed in this dissertation is applied to the design of a space-based virtual telescope comprised of two spacecraft flying in formation. A reformulation of the orbit dynamics of the spacecraft is found to avoid the introduction of truncation errors due to tight formation constraints that render solving the optimization problem impossible. A sequential approach to applying MDO to the design of a space-based virtual telescope is also found to be more robust than solving the MDO problem where all disicplines are considered simultaneously.

Chapter 1

Introduction

1.1 Motivation

The field of MDO encompasses the application of optimization techniques to solve complex design problems involving multiple engineering disciplines such as structures, aerodynamics, vehicle dynamics, and propulsion. In today's rapidly evolving landscape, multidisciplinary design optimization (MDO) has found its application in diverse industries such as aerospace, automotive, and wind farm optimization. These systems involve multiple disciplines that must all be considered in the design process. Increasingly demanding requirements for such systems lead to increasingly complex systems, and as the complexity of systems continues to increase, the demand for robustness, reliability, and the consideration of uncertainty pose significant challenges for MDO practitioners. In this context, the evaluation of models of large-scale systems plays a crucial role in gaining valuable insights that aid the design process. However, implementing the necessary software for finding an optimal design for large-scale systems introduces complexity within the design process, often obscuring the original model description and making it challenging to understand and maintain software used for simulation. Factors such as model composition, two-way coupling, and derivative computation contribute to this complexity. The complexity of these systems also poses a challenge to applying optimization algorithms to solve large-scale MDO problems. Manually implementing software that evaluates the model of the system within the optimization process exacerbates these challenges, limiting

the model developer's ability to explore the design space and compromising the potential for discovering innovative solutions. Thus, the reliance on manual implementation of software that evaluates the model within the optimization process not only amplifies the challenges faced by MDO practitioners but also impedes the overall efficiency and effectiveness of the design optimization process.

Evaluating a model involves simulating the behavior of the system based on the model's description. The description of a model can be interpreted as a specification of a program that simulates a system's behavior. Consequently, the software specification must be sufficiently descriptive to accurately represent the physical system modeled. There is nothing about the physical system itself that imposes restrictions on how the physical system is modeled. As pointed out in [2] (citing [3]),

> ... [W]hile the species in the world obeys rules, ... whether formal or mathematical, no 'rule' whatsoever can dictate how one should map the hypothesized rules in the world onto the rules in the model. In the words of Rosen, while the world and the model are each internally 'entailed', nothing entails the world with the model.

To use the terminology in the quote above, it is the responsibility of the engineer to define the rules in the model that are consistent with the hypothesized rules in the world. The rules in the model specify how software that is meant to accurately simulate the physical system behaves. Often, the role of an engineer involves specifying the rules in the model and ensuring that they are a faithful representation of the physical system, as well as ensuring that the implementation of the software satisfies the specification. Ideally, MDO practitioners should concentrate their efforts on specifying the rules in the model and ensuring that they are a faithful representation of the physical system rather than ensuring that the implementation of the software satisfies the specification.

Developing such programs can be error-prone since the source code does not clearly differentiate between the "what" (specification) and the "how" (implementation). It is this later portion of the MDO workflow that this dissertation aims to automate for MDO practitioners,

separating the model description (program specification) from the program implementation. Although there are existing solutions to address these challenges, they still have inherent limitations that can be overcome through a paradigm shift.

Recognizing the need for a transformative approach, I propose a new paradigm that emphasizes the integration of automated processes into distinct components within an overarching methodology. This paradigm shift forms the foundation for a novel approach to MDO that separates the manual implementation of low-level software from the original model description. By adopting this approach, MDO practitioners can streamline their workflows, enhance optimization capabilities, and gain a deeper understanding of the design space.

This dissertation builds upon computational architectures [4] and software frameworks [5, 6, 7, 8] for MDO, leveraging the Computational System Design Language (CSDL) and its compiler (Chapter 3). Another obstacle in successful MDO lies in constructing mathematical models amenable to optimization, which this dissertation explores in a study (Chapter 5). The language and compiler work in tandem to separate the model description from the computational architecture.

1.2 Research Needs and Objectives

Motivated by the challenges MDO practitioners face, I have identified the following critical research needs:

- The need to automate efficient and accurate sensitivity analysis for the class of problems encountered in MDO.
- The need to remove obstacles to implementation of software that simulates the behavior of a system for a given model description
- The need to gain insight into the properties of the physical system's behavior as well as the program's run-time behavior from the structure of the model description.

- The need to reformulate model descriptions to mitigate numerical difficulties encountered in specific MDO problems.

Based on these needs, the objectives of this dissertation are to:

- Apply a three-stage compiler design to the process of automatically performing efficient and accurate sensitivity analysis for the class of models present in MDO problems.
- Measure the effect of using a declarative paradigm to describe models used in MDO problems on user code complexity.
- Measure the effect of the model structure on the run-time memory complexity of the software used to evaluate the model.
- Apply MDO to the problem of designing a space-based virtual telescope using a model description amenable to gradient-based optimization.

In pursuit of these research objectives, I have arrived at the following thesis statement:

> The use of graph representations in multidisciplinary design optimization enables efficient and accurate sensitivity analysis and compile-time complexity analysis, accelerating and enhancing the development of models and software implementations used within optimization frameworks.

To support this thesis, I provide the foundation for generating software used to evaluate models of physical systems using graph representations, enabling efficient and accurate sensitivity analysis, and separating the model description from implementation of the software used to evaluate the model. The methodology developed in this dissertation has been applied to a variety of MDO problems [9, 10, 11, 12, 13, 14, 15, 16].

The separation of model description from implementation of the software used to evaluate the model brings to light a set of challenges that directly arise from the choice of the model description itself, rather than the specific implementation details of the software used to evaluate the model. Recognizing this gap, I explore a specific challenge within the context of a virtual

telescope featuring tight formation constraints. In this scenario, the large difference in magnitudes between the constraint values and the intermediate variables in the model makes gradient-based optimization impossible due to insufficient numerical precision. Model descriptions, viewed as program specifications, inherently require that if a solution to the optimization problem exists, the gradient-based optimizer can converge to a solution with the necessary level of precision. Consequently, model developers are tasked with providing model descriptions that serve as program specifications whose implementations are capable of satisfying this requirement. Importantly, these challenges are distinct from the implementation details of the software used to evaluate the model and are an integral part of the model description process. While this stage of the MDO workflow remains a manual process, it is more distinctly isolated from the overall MDO workflow.

1.3 Dissertation Roadmap

This dissertation is organized as follows:

- Chapter 2 presents related work and background information for chapters 3, 4, and 5.
- Chapter 3 presents a graph-based methodology for constructing the software used to evaluate the model and automates adjoint-based sensitivity analysis. The methodology centers around a graph representation of an MDO problem. The abstract representation can be used to inform the implementation of a software implementation that simulates the behavior of the system. The methodology and accompanying analyses presented in Chapter 4 naturally lend themselves to be implemented as a three-stage compiler dedicated to building executable objects that simulate the behavior of models used in MDO problems. Models present in MDO problems also lend themselves to be described using a combination of the functional programming paradigm and object-oriented style. Collaborators and I designed the Computational System Design Language (CSDL) as a functional programming language with an object-oriented style to meet this requirement.

- Chapter 4 presents analyses that can be performed on the graph representation itself that can also give greater insight into properties of the software implementation, or of the system itself. These analyses are built into the three-stage compiler presented in Chapter 3 and provide users with the ability to measure potential performance and memory complexity for a given model.

- Chapter 5 explores the challenge of describing a system using a model amenable to gradient-based optimization through the application of MDO to the design and operation of a virtual-telescope spacecraft swarm in orbit about the Earth. Spacecraft swarms are an attractive option for space-based telescopes because their mass scales well with the size of the telescope. That is, large telescopes do not have to be manufactured, but can still be deployed. The model used in this study is defined in CSDL using the methodology presented in Chapter 3. The main challenge in generating this contribution is satisfying tight formation requirements. The orbit model uses the positions of the spacecraft relative to the center of the Earth, which are on the order of thousands of kilometers, to define the formation. These requirements are cast as constraints in the optimization problem on the order of tens of millimeters. The constraint values are smaller than the intermediate states by eight orders of magnitude. As a consequence, these constraints cannot be satisfied using an orbit model that models the states of the spacecraft relative to the center of the Earth. This is due to the lack of numerical precision available on a 64-bit computer to update the design variables to update the trajectory of each spacecraft to satisfy the constraints. Thus, the problem requires models for spacecraft proximity or formation dynamics. A new orbit model is developed for this problem that uses the positions of the spacecraft relative to a reference orbit, reducing the relative order of magnitude from 8 to 4. For this particular problem, choice of the model description defines a scalable approach for spacecraft swarm design optimization.

At the core of this research lies the recognition that software plays a pivotal role in

analyzing systems by evaluating their models. This evaluation process is a critical aspect of the design process. To facilitate iterative design improvements, optimization frameworks have been developed, accelerating the evaluation of models of large-scale systems.

Chapter 2

Related Work

2.1 Multidisciplinary Design Optimization

This chapter provides an overview of MDO and its practical applications. MDO is an approach that aims to optimize complex engineering systems by considering multiple disciplines simultaneously [17, 18]. By integrating and coordinating the design of various subsystems or disciplines, such as structures, aerodynamics, controls, and more, MDO seeks to achieve an optimal overall system design.

MDO has found successful applications in various industries and fields. In aerospace, MDO has been used to optimize commercial aircraft route allocation [19], electric vertical takeoff and landing (eVTOL) vehicles [20, 21], and spacecraft [22, 23, 16].

Gradient-free methods, also known as derivative-free methods, are optimization techniques that do not rely on gradient information of the objective and constraint functions. These methods can be useful when the objective and constraint functions are non-differentiable or when the gradient information is costly or unavailable. Gradient-based methods, on the other hand, utilize gradient information (first-order derivatives) of the objective and constraint functions to guide the search for optimal solutions. When available, gradient information provides gradient-based methods with several advantages over gradient-free methods. Gradient-based methods can converge to an optimal solution faster than gradient-free methods, especially when the design space is large or when the number of design variables is high. Gradient-based methods can

handle large-scale optimization problems with a large number of design variables and constraints more effectively.

Gradient-based methods are not without their obstacles, however. Non-differentiability of the objective or constraint functions is a major obstacle to applying gradient-based methods to optimization problems. In some cases, certain functions can be approximated by a continuous and differentiable function. Discrete design variables also pose a challenge because a function cannot be differentiated with respect to a discrete variable. In the case where discrete variables are integers, the problem is a mixed-integer problem. When the problem is differentiable in the design variables, computing gradients can also be computationally expensive, especially when the objective and constraint functions require complex simulations or numerical evaluations. Finally, gradient-based methods may converge to local optima instead of the global optimum.

2.2 Review of Derivative Computation Methods

Derivative computation is a fundamental aspect of gradient-based optimization and sensitivity analysis, and various methods have been developed to compute exact derivatives. This section provides a comprehensive review of derivative computation methods, starting with an overview of numerical methods that require less implementation effort but they do not provide exact derivatives. The focus then shifts to analytic methods, which compute exact derivatives but involve manual differentiation and implementation procedures. The direct method and adjoint methods compute derivatives of composite functions. The adjoint method offers performance advantages over the direct method when the number of design variables is large compared to the number of constraints and the objective. Symbolic differentiation and automatic differentiation (AD) are presented as alternative approaches for computing exact derivatives. All of the methods here can be derived from the unifying derivative equation [4], which is reviewed here as well.

Numerical methods require the least amount of implementation effort of all the methods reviewed here. Among numerical methods, finite-difference methods require the least amount

of implementation effort. Finite-difference methods approximate derivatives by perturbing the function in each direction by a finite amount and using that perturbation to compute the change in the value of the function with respect to each of its inputs. Although finite-difference methods are straightforward to implement, they suffer from truncation error proportional to the magnitude of the perturbation. As the size of the perturbation decreases past a certain point, finite-difference methods also suffer from round-off/subtractive cancellation error. Minimizing error requires choosing an optimal step/perturbation size that is dependent on the function.

The complex-step method on the other hand, perturbs the function by a step along the imaginary axis [24, 25]. As a result, the complex-step does not suffer from round-off/subtractive cancellation error, and the size of the perturbation can be made arbitrarily small (but not zero). These methods, however, do not compute exact derivatives, so other methods have been developed to compute derivatives. Furthermore, neither finite-difference nor complex-step methods are efficient methods for derivative computation as they require at least n function evaluations to compute a gradient.

In contrast to numerical methods of differentiation, the methods that follow all compute exact derivatives. Hand coding analytic derivatives instead of relying on numerical approximations results in a program computing exact derivatives. Although analytic methods compute exact derivatives and can be efficient, the process of manually implementing derivatives in this way is tedious and error-prone, especially for large systems.

In general, computing exact derivatives is a two-step process: first, the partial derivatives of component functions must be computed, and then those results are used to solve for the total derivatives of the composite functions for the objective and constraints with respect to the design variables [26]. When implementing analytic derivatives, the process can be automated somewhat by requiring users to provide partial derivatives for component functions and then automatically applying the chain rule to compute the total derivatives. For instance, let $f \in \mathbb{R}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, and $F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$. In the case where y is defined explicitly in terms of x, the (forward) chain

rule is applied to $f = F(x,y)$ as:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x}. \tag{2.1}$$

In the case where y is implicitly defined by $R(x,y) = 0$, where $R : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$, and does not have a closed-form solution, an iterative solver is used to converge a residual $r = R(x,y)$, where $r \in \mathbb{R}^m$. When the model of the system is evaluated, $R(x,y) = 0$, and the derivative $\mathrm{d}r/\mathrm{d}x$ is given by

$$\frac{\mathrm{d}r}{\mathrm{d}x} = \frac{\partial R}{\partial x} + \frac{\partial R}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x} = 0. \tag{2.2}$$

If $\partial R/\partial y$ has a well-defined inverse, then the implicit function theorem can be applied to solve for $\mathrm{d}y/\mathrm{d}x$ according to the adjoint equation [26, 4],

$$\frac{\partial R}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x} = -\frac{\partial R}{\partial x}. \tag{2.3}$$

Note that an explicit function $y = Y(x)$ need not be known to compute $\mathrm{d}y/\mathrm{d}x$ at any point $r = R(x,y)$ to solve (2.3). Solving (2.3) for $\mathrm{d}y/\mathrm{d}x$ and substituting into Equation (2.1) gives

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \frac{\partial F}{\partial y}\frac{\partial R}{\partial y}^{-1}\frac{\partial R}{\partial x}. \tag{2.4}$$

The adjoint method is given by

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} + \psi^T\frac{\partial R}{\partial x}, \tag{2.5}$$

where

$$\psi^T = -\frac{\partial F}{\partial y}\frac{\partial R}{\partial y}^{-1} \tag{2.6}$$

is called the adjoint vector. The adjoint vector can be obtained by solving

$$\frac{\partial R}{\partial y}^T \psi = -\frac{\partial F}{\partial y}^T . \tag{2.7}$$

Computing derivatives via Equation (2.1) and (2.5) are referred to as the *direct method* and the *adjoint method*, respectively, and both equations generalize to cases where f is a vector [26, 4]. In the case where f is a vector, ψ is referred to as the adjoint *matrix*. Some authors make the distinction between the adjoint method (2.5) and the *coupled* adjoint method depending on whether or not $r = R(x,y)$ has a closed-form solution; i.e., whether or not the system is coupled [27]. Although the direct and adjoint methods are referred to as analytic [4] and semi-analytic methods [27] in the MDO literature, the partial derivatives of the component functions within the system need not be computed using any particular method of differentiation of the methods outlined here [4].

The direct method and the adjoint method are also called forward-mode and reverse-mode differentiation, respectively. Forward-mode and reverse-mode differentiation are also referred to as the forward chain rule and reverse chain rule, respectively [26]. Forward-mode differentation is more efficient when the number of constraints and the objective is greater than the number of design variables, and reverse-mode differentiation is more efficient when the number of design variables is greater than the number of constraints and the objective [4, 5, 7].

Fully automated methods of derivative computation are also available. Symbolic differentiation automatically generates expressions for derivatives of the expressions in the model, computes exact derivatives, and can be very efficient. The main disadvantage of symbolic differentiation packages is that they cannot perform implicit differentiation, thus limiting their versatility [28, 29].

Automatic Differentiation (AD) is similar to symbolic differentiation and has some similarities to the complex-step method [24, 25]. Some AD libraries construct a graph of vectorized operations, reducing the size of the graph and thus the memory footprint of the

library [30, 31, 32, 33]. Some AD libraries implement forward-mode differentiation (equivalent to the direct method) only, and some implement reverse-mode differentiation (equivalent to the adjoint method) [30, 31, 32, 28, 29, 34]. AD libraries are used within the machine learning community for backpropagation (equivalent to applying reverse-mode differentiation) when generating neural networks [35]. Some AD libraries implement implicit differentiation by a *trace method*, which unrolls the loops and stores all iterations computed during the course of solving implicit functions/nonlinear systems of equations and applies explicit differentiation using either forward-mode or reverse-mode differentiation [29]. Some AD libraries implement vectorization, reverse-mode differentiation, and implicit differentiation [30, 31, 32, 28, 29, 34].

All discrete methods of computing derivatives can be derived from the unifying derivative equation (UDE) [4], originally referred to as the unifying chain rule [26]. The UDE is given by

$$\frac{\partial \mathscr{R}}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}r} = \mathscr{I} = \frac{\partial \mathscr{R}}{\partial u}^{T}\frac{\mathrm{d}u}{\mathrm{d}r}^{T}, \tag{2.8}$$

where

$$u = \begin{pmatrix} x \\ y \\ f \end{pmatrix}, \text{ and } \mathscr{R}(u) = \begin{pmatrix} x - x^{*} \\ -R(x,y) \\ f - F(x,y) \end{pmatrix} \tag{2.9}$$

is the system cast as a nonlinear system of equations called the *fundamental system* [4]. Solving the left-hand side of Equation (2.8) corresponds to forward-mode differentiation while solving the right-hand side corresponds to reverse-mode differentiation. Forward-mode and reverse-mode differentiation correspond to the direct method and adjoint method, respectively [4].

Although the values of the total derivatives are the solution to the linear system (2.8), AD libraries that implement implicit differentiation do not store any matrices or compute a solution to Equation (2.8) by solving a system of linear equations.

So far, these challenges have been addressed within the MDO community by designing

computational architectures [4] and software frameworks [5, 7]. OpenMDAO implements the modular analysis and unified derivatives (MAUD) architecture [4] and computes total derivatives by solving the UDE (2.8), automatically choosing forward-mode or reverse-mode differentiation [5, 7].

In the broader engineering community, algebraic modeling languages (AMLs) [36, 37] have provided model developers with interfaces to describe mathematical models without the additional overhead of implementing software [6, 8, 38, 39, 40, 41, 42]. The differences between these AMLs are discussed in detail in Section 2.5.

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.

2.3 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 [43] and [44]. 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 dissertation, 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 [43].

Modeling paradigms can be separated into graph-based and language-based paradigms [44][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 [43], declarative paradigms are referred to as constraint paradigms). Figure 2.1 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 [44], 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 [43]. The object-oriented paradigm associates functions with a shared state. Modeling languages like ObjectMath [45], Modelica [46], Pyomo [6, 8] and the *ad hoc* langauge defined by the MDO framework OpenMDAO [5, 7] 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-

[1] The term "graph-based" here refers to a property of languages, and not a property of the methodology presented in this dissertation.

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 [47]. 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 [45] and Modelica [46].

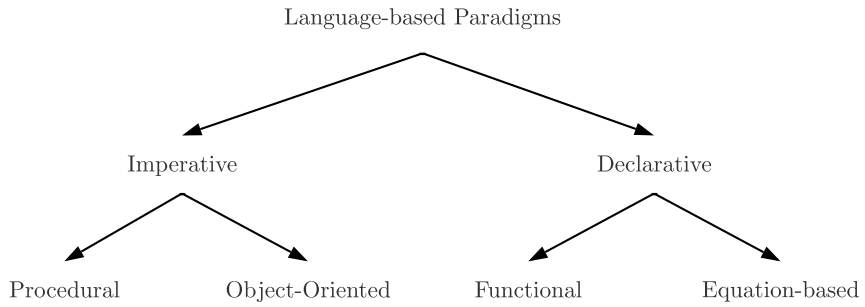


Figure 2.1. Diagram showing the relationship between language-based paradigms. Language paradigms can be broadly separated into imperative and declarative paradigms. The imperative family of paradigms contains procedural and object-oriented paradigms, and the declarative paradigm contains the functional 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 [38] inherits the multiple paradigms of the Julia language [48], a high-performance dynamic programming language with support for automatic differentiation [49, 35] 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 Section 3.4, the Computational System Design Language (CSDL), uses a functional paradigm to express relationships between variables, and an object-oriented paradigm to compose models hierarchically.

2.4 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 2.3 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) [47] and compile programs using either the FEniCS Form Compiler (FFC) [50, 51, 52, 53] or the SyFi Form Compiler (SFC) [54, 55].

2.5 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 [39, 40, 41], 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 [6, 8] is an AML that provides semantics for hierarchical composition of models. Optimica [42], 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 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 [4], which enables the use of hierarchical solvers to evaluate the model, and automatically solves the unifying derivative equation (UDE) [26] to compute total derivatives, overcoming the curse of dimensionality.

None of the modeling languages presented here enables 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 Chapter 3.

[2]There 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.

2.6 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 3.4, is a functional and object-oriented 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).

Chapter 2, in part, has been submitted for publication of the material as it may appear in Structural and Multidisciplinary Optimization 2023. Gandarillas, Victor; Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT, Springer Nature, 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 3

A Graph-based Methodology for Constructing Computational Models that Automates Adjoint-based Sensitivity Analysis

3.1 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 [56].

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 AMLs [36, 37], 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 prebuilt solvers. A number of AMLs [6, 8, 38, 39, 40, 41, 42] have been developed with various overlapping features, but they do not completely eliminate manual or *ad hoc* processes.

The focus of this chapter is to automate manual and *ad hoc* processes used to develop computational models for multidisciplinary systems, and also present a new AML that hides the implementation details accordingly. I propose a graph-based methodology for constructing computational models for large-scale multidisciplinary design optimization problems to enable accurate and efficient adjoint sensitivity analysis. I implement the methodology as a three-stage compiler and develop a new modeling language called the Computational System Design Language (CSDL) that provides an appropriate input to the compiler front end that works well with multidisciplinary models. At the heart of the three-stage compiler is an implementation-agnostic graph representation that enables functionality beyond derivative computation at both compile time and run time, such as time and space complexity analysis, parallel computing, efficient memory management, and efficient uncertainty quantification.

The chapter is organized as follows. Section 3.2 presents the graph-based methodology for constructing computational models and the design of the compiler that implements the new methodology. Section 3.3 presents model code optimizations applied to the graph representation of Section 3.2.2. Section 3.4 presents an overview of the new algebraic modeling language (AML), CSDL and provides resources for the reader to access CSDL and compiler source code for developing numerical and computational models. Section 5.6 presents a comparison of lines of code using CSDL versus OpenMDAO. Finally, Section 3.6 presents the conclusions of this work along with future work.

3.2 Model Code Generation

This section presents a novel graph-based methodology for building computational models that forms the contribution of the chapter. 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 CSDL, described in Section 3.4.

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 as a *Simulator*[1], which is the name used in CSDL. Figure 3.1 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.

[1] This is the same as the concept of a model within the `Problem` class in OpenMDAO

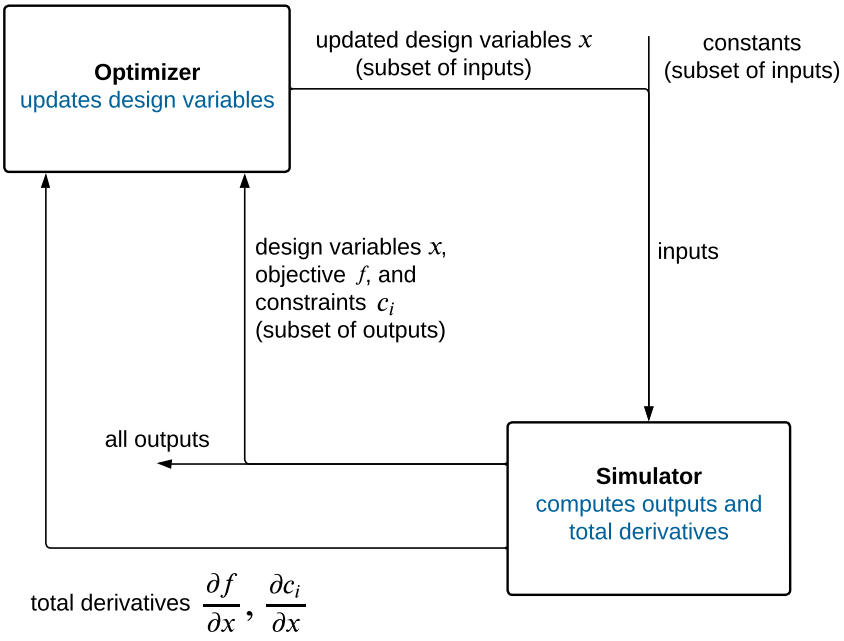


Figure 3.1. The compiler constructs a Simulator object that evaluates the computational model within an optimization framework. The Simulator and Optimizer execute sequentially until the Optimizer converges to an optimal solution.

3.2.1 Compiler Overview

The methodology presented in this chapter is formulated as a three-stage compiler that constructs the Simulator object in Figure 3.1. Figure 3.2 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 [57]. The IR that the compiler in this chapter 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* $(e_1, e_2, ..., e_{n-1})$ *for which there is a sequence of nodes* $(v_1, v_2, ..., v_n)$ *such that* $e_i = (v_i, v_{i+1})$ *for* $i = 1, 2, ..., n-1$.

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

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 3.2 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 $f : \mathbb{R}^n \to \mathbb{R}^m$ is a numerical model and $\mathscr{M}$ is the set of possible models f, a compiler front end can be interpreted as a function $p : \mathscr{M} \to \mathscr{D}$ that generates a DAG, $G \in \mathscr{D}$, where $\mathscr{D}$ is the set of all DAGs.

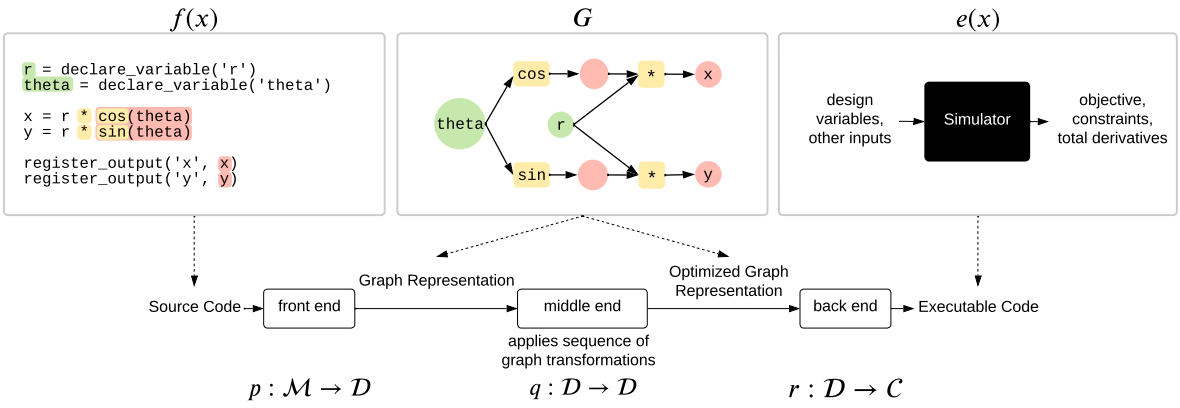


Figure 3.2. A three-stage compiler translates a numerical model to executable code implementing the computational model. The user supplies source code and the compiler generates a *Simulator* that evaluates the computational model.

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 $q : \mathscr{D} \to \mathscr{D}$ that transforms the graph representation $G \in \mathscr{D}$ of a model $f \in \mathscr{M}$ 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 $r : \mathscr{D} \to \mathscr{C}$, where $\mathscr{C}$ 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 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 3.4), to a graph representation described in Section 3.2.2.

3.2.2 Graph Representation of Models

The graph representation is an intermediate representation (IR) with a directed ayclic 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 3.2. 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 3.3) 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 Chapter 4 and the capability to perform these analyses, such as those shown in Section 4.6.

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 [47]. 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 3.3.

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.

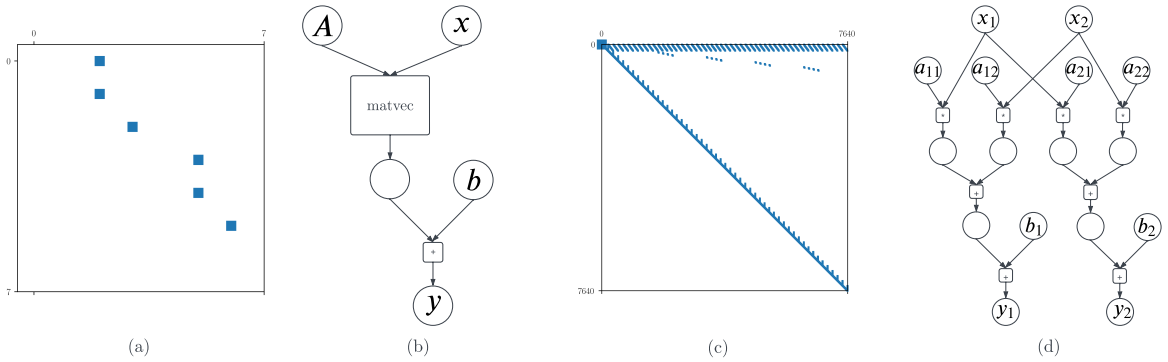


Figure 3.3. Adjacency matrices and graph representations of $y = Ax + b$, showing that the size of the variables in a numerical model has no effect on either the size or the order of the graph representation. (a) Adjacency matrix sparsity pattern for graph representations generated by compiler front 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 3.4 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.

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 3.5 shows the adjacency matrices

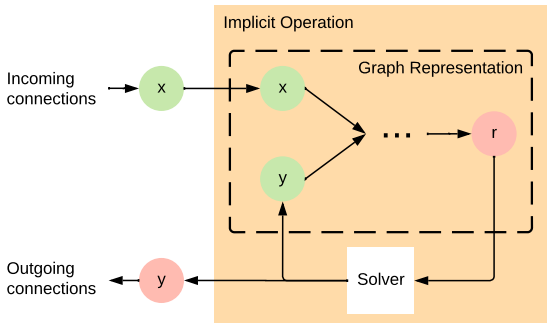


Figure 3.4. An implicit operation defined by a model with known variable x, unknown variable y, and residual r. Users provide a model definition, and a choice of a root-finding solver, but do not provide implementation details other than solver options, e.g., maximum number of iterations.

for a model with two implicit operation nodes.

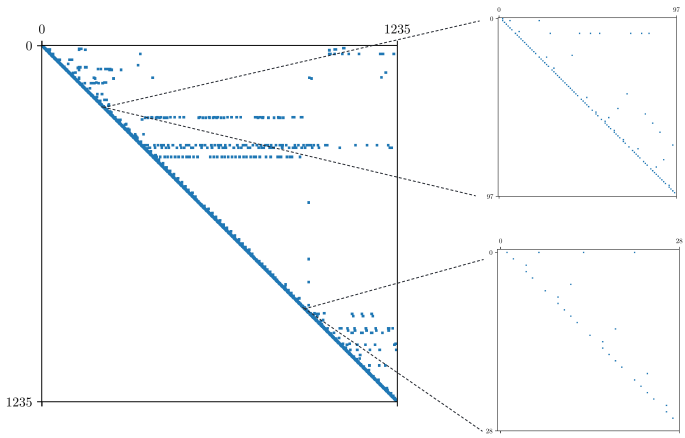


Figure 3.5. Adjacency matrix for a model of a propeller rotor using the blade element momentum-based ideal-loading design (BILD) method with two implicit operations, along with an adjacency matrix for each model defining each implicit operation.

3.2.3 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. MDO models have particular structure and characteristics 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 3.2.2) also appear more often in MDO models due to coupling between disciplines. A compiler back end uses the graph representation in Section 3.2.2 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 implementation.[2] The Computational System Design Language (Section 3.4) provides custom operations for this purpose.

Although this chapter does not present an automatic differentiation (AD) library, it 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 3.2.2, prior to computing exact derivatives [35, 28]. Just as a compiler back end in the three-stage compiler presented in Section 3.2.1 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 [58, 59, 60]. 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

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

library.

Implicit operation nodes (Figure 3.4) encode the relationship $r = R(x, y)$ between a residual r, and a known variable x and unknown variable y. The adjoint method is the result of computing $\partial R/\partial x$, $\partial R/\partial y$ and applying the implicit function theorem under the condition that $\partial R/\partial y$ has an inverse. This strategy of implicit differentiation is outlined in [28, 29, 34] and also implemented in AD libraries in Julia [61],[3] Python [34, 32], and C++ [30, 31]. Although this strategy of implicit differentiation is implemented in existing AD libraries, the nested nature of the graph representation enables explicit differentiation of each graph representation; i.e., the $\partial r/\partial x$, $\partial r/\partial y$ can be computed explicitly for the graph representation within each implicit operation (Figure 3.4) using any differentiation method, including an AD library that only supports explicit differentiation.

Although the graph representation, JAX [34, 32], and Stan-math [30, 31] represent implicit relationships in a similar way, JAX and Stan-math are dedicated AD libraries, while the graph representation in Section 3.2.2 is not limited to derivative computation (See Section 3.2.4).

3.2.4 Beyond Derivative Computation

The three-stage compiler provides benefits for MDO that extend beyond derivative computation. The implementation-agnostic graph representation (Section 3.2.2) used in the three-stage compiler also enables functionality at both compile time and run time; e.g., time and space complexity analysis (Chapter 4), parallel computing, efficient memory management, and efficient uncertainty quantification.

Since the compiler uses an implementation-agnostic graph representation, generated code can offer better flexibility, compatibility, and performance of MDO model evaluation over existing frameworks by targeting specific hardware architectures. Because the compiler is free

[3] Another AD library that supports implicit differentiation is avaialable at https://gdalle.github.io/ImplicitDifferentiation.jl/

to generate code for any architecture, simulation code could potentially run on an embedded system[4].

Chapter 4 presents time and space complexity analysis performed at compile time using the graph representation. A compiler back end can use to generate a computational model that achieves the gains presented in Chapter 4 to parallelize computation and manage memory efficiently.

In some cases, MDO models also contain uncertain inputs. When uncertain inputs are present in the model, optimizations can be applied to the graph representation to reduce the number of model evaluations [10, 13, 62], regardless of the compiler back end's implementation strategy.

3.3 Model Code Optimizations

The graph representation in Section 3.2.2 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.

[4]This assumes the existence of a compiler back end that can support code generation for a particular architecture, which is not the case at the time of writing.

3.3.1 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 [63] 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.

By traversing the nodes in S and $\{w : (v,w) \in \mathscr{E}\}$ for each $v \in \mathscr{N}$ in the right order, variable allocation may be delayed as much s possible. A function $u : \mathscr{D} \times \mathscr{N} \to \mathbb{N}$ 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 $\{w : (v,w) \in \mathscr{E}\}$ for each $v \in \mathscr{N}$. 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 $v \in \mathscr{N}$, if the nodes in $\{w : (v,w) \in \mathscr{E}\}$ are initially sorted by $u(G,w)$ in descending order, then the nodes initially stored in $\{w : (v,w) \in \mathscr{E}\}$ with the same outdegree $\deg^+(w)$ prior to applying Kahn's algorithm

Algorithm 1. Kahn's Algorihm

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

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.

3.3.2 Dead Code Removal

Not all user code in the numerical model is necessarily translated to the computational model. 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 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 [63] 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.

3.3.3 Combining Element-wise Operations

Many standard functions are vector-valued functions of the form $f : \mathbb{R}^n \to \mathbb{R}^n$, 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

$$\sum_{i=1}^{m} p^{(i)} + (m-1)n, \tag{3.1}$$

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 [24]—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:

$$\frac{\partial f}{\partial x} = \text{diag}\left(\frac{\text{Imag}[f(x+ih\mathbb{1}_n)]}{h}\right), \tag{3.2}$$

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 $f : \mathbb{R}^n \to \mathbb{R}^n$ 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 $x+ih\mathbb{1}_n$ and dividing the n elements of $\text{Imag}[f(x+ih\mathbb{1}_n)]$ 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, $p^{(i)} = n, \forall i \in \{1,\ldots,m\}$.

Then from Equation (3.1),

$$mn + (m-1)n \geq mn + 2n, \tag{3.3}$$

$$mn + mn - n \geq mn + 2n, \tag{3.4}$$

$$mn \geq 3n, \text{ and} \tag{3.5}$$

$$m \geq 3. \tag{3.6}$$

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 3.2.2) to identify paths containing operation nodes representing only functions with diagonal Jacobians and combines them into a single operation representing the composite function with diagonal Jacobian.

3.4 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 3.2. 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 3.1 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 3.1. 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 3.2), as well as functions

or compositions of functions (as shown in Listing 3.3). 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 3.2. 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 3.3. 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 3.2.2. The `GraphRepresentation` class constructor serves as the compiler front end, which constructs the DAG from an instance of a user-defined `Model` sublcass. 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 3.3), 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 in-

stance 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.

`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

[5] Back 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.

whose residuals cannot be defined using a composition of functions from the standard library.

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.

3.5 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 [64, 65], and `lsdo_bemt` [21]. 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 3.6 shows the difference in the number of lines of code used to define the models in each project.

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.

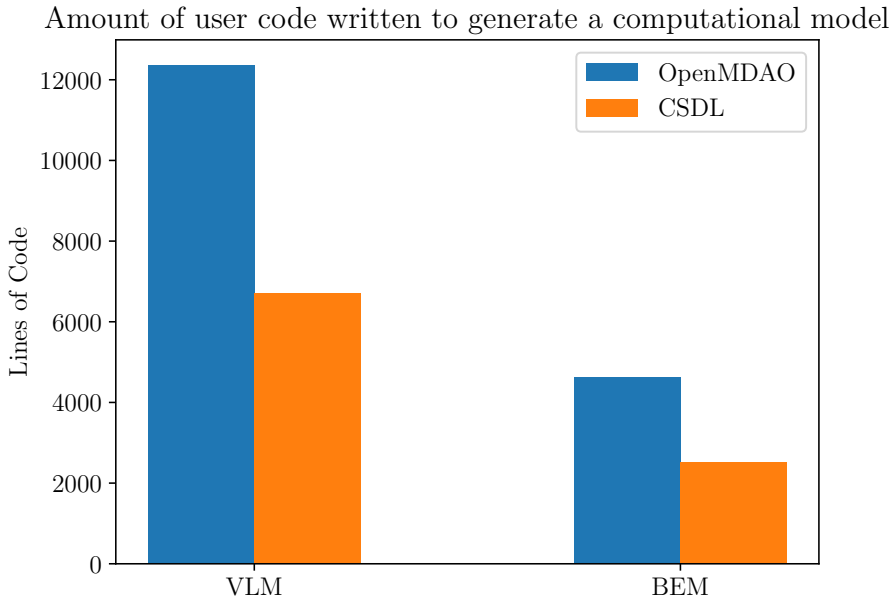


Figure 3.6. The number of lines of code required to implement or generate a computational model using OpenMDAO is ∼2x greater than that of using CSDL. OpenMDAO is a software framework that uses the current state-of-the-art method for exact scalable sensitivity analysis of complex models.

3.6 Conclusion

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 chapter 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 AML called 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$2x.

3.7 Acknowledgments

Special thanks are given to Marius Ruh, Jiayao Yan, and Luca Scotzionovsky for helping generate the data in Figure 4.3. This work was supported in part by the National Science Foundation under grant no. 1936557 and NASA grant number 80NSSC21M0070.

Chapter 3, in full, has been submitted for publication of the material as it may appear in Structural and Multidisciplinary Optimization 2023. Gandarillas, Victor; Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT, Springer Nature, 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 4

An Approach to Measuring Computational Costs Using Graph Representations of Models

This chapter presents analyses of the time and memory costs associated with generating and evaluating a computational model, using the compiler graph representation presented in Section 3.2.2. 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 that 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. Chapter 4 presents an analysis of the run-time and compile-time costs incurred by the compiler that implements the methodology presented in Section 3.2. Numerical results demonstrating the potential for improved run-time costs are shown in Section 4.6. This chapter also includes complexity analyses for evaluating computational models and their derivatives.

4.1 Nomenclature

The compiler front end generates a DAG representing a model, denoted by $G(\mathscr{N},\mathscr{E})$ (Definition 4), where $\mathscr{N}$ is the set representing nodes, which is defined by the union of the variables $\mathscr{V} \subset \mathscr{N}$ and operations $\mathscr{O} \subset \mathscr{N}$, and $\mathscr{E} = \{(u,v) : u \in \mathscr{V} \Leftrightarrow v \in \mathscr{O}, u \in \mathscr{O} \Leftrightarrow v \in \mathscr{V}\}$ is the set of directed edges representing dependencies between variables and operations. The graph G stores each variable node $v \in \mathscr{V}$, in general representing a multidimensional array, and each operation node $o \in \mathscr{O}$, 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 $x \in \mathscr{N}$ is zero if $x \in \mathscr{O}$. If $x \in \mathscr{V}$, 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 $o \in \mathscr{O}$. A function $h : \mathscr{O} \to \mathscr{N}^n$ 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.

4.2 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

$$\gamma_{\text{rev}}(x,k) = \sum_{i=1}^{n} \text{size}\left(x_i\right). \tag{4.1}$$

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

$$f(x,k) = \begin{cases} \sum_{i=1}^{k-1} \text{size}(x_i)\text{sgn}\left(\sum_{j=k}^{n} A_{ij}\right) + \sum_{i=k}^{n} \text{size}(x_i)A_{ki} & x_k \in \mathscr{O} \\ 0 & x_k \in \mathscr{V} \end{cases}, \tag{4.2}$$

where $x = (x_1, \ldots, x_n)$ is a vector of $n = |\mathscr{N}|$ nodes with $x_i \in \mathscr{N}, i \in \{1, \ldots, n\}$, $k \in \mathbb{N}$, 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 $x_k \in \mathscr{O}$ 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 $x_i \in \mathscr{V}$ within the sums in Equation (4.2) contribute to the memory required for computing $x_k \in \mathscr{O}$, and not $x_k \in \mathscr{O}$ itself. In the case where $x_k \in \mathscr{O}$, the first term, $\sum_{i=1}^{k-1} \text{size}(x_i)\text{sgn}\left(\sum_{j=k}^{n} A_{ij}\right)$, takes into account the memory that must be allocated for all variables on which current and future operations depend must be allocated while operation $x_k \in \mathscr{O}$ 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 $x_k \in \mathscr{O}$ 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 $o \in \mathscr{I}$, where $\mathscr{I} \subseteq \mathscr{O}$ is the set of implicit operations contained in the model, described in Section 3.2.2. Taking into account the additional memory cost incurred by the presence of implicit operations, the total memory cost is given by the recursive function

$$\gamma_{\text{fwd}}(o) = \max\left(\max_{k \in \{1,\ldots,n\}} (f(h(o),k)), \max_{(q,k) \in Q} (f(q,k) + \gamma_{\text{fwd}}(q))\right), \tag{4.3}$$

where $o \in \mathscr{O}$, $Q = \{(q_k,k) : q_k \in \mathscr{N}^n \cap \mathscr{I}\}$ 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.

4.3 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

$$\alpha_{\text{dense}}(x,k) = \sum_{a=1}^{k-1} \sum_{b=k+1}^{n} \text{size}(x_a)\text{size}(x_b)\text{sgn}\left(\sum_{j=k}^{n} A_{ak}\right)\text{sgn}\left(\sum_{j=k}^{n} A_{kb}\right). \tag{4.4}$$

The right-hand side of (4.4) 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, 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,

$$\beta_{\text{rev}}(o) = \sum_{k=1}^{n} \alpha(h(o),k) + \sum_{i \in \mathscr{I}} \beta_{\text{rev}}(i). \tag{4.5}$$

This represents the best-case memory cost of computing total derivatives for the system-level model, assuming that reverse-mode differentiation is used for the system-level model and all models are 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 the k^{th} block of the total derivatives when using forward-mode differentiation is given by

$$p(x,k) = \begin{cases} \sum_{i=1}^{k-1} \alpha(x,i) \text{sgn}\left(\sum_{j=k+1}^{n} A_{ij}\right) + \alpha(x,k) & x_k \in \mathscr{O} \\ 0 & x_k \in \mathscr{V} \end{cases}, \tag{4.6}$$

where $\alpha(x,i)$ represents 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 $x_k \in \mathscr{O}$. If x_k is a variable, p is zero. In the case where $x_k \in \mathscr{O}$,

the first term, $\sum_{i=1}^{k-1} \alpha(x,i)\text{sgn}\left(\sum_{j=k+1}^{n} A_{ij}\right)$, takes into account all partial derivatives on which current and future components of the total derivatives depend must be allocated while operation $x_k \in \mathscr{O}$ is executed. The second term, $\alpha(x,k)$, takes into account the memory required for computing the partial derivatives of the current operation $x_k \in \mathscr{O}$.

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,

$$\beta_{\text{fwd}}(o) = \max(\max_{k\in\{1,\ldots,n\}}(p(h(o),k)), \max_{(q,k)\in Q}(p(q,k)+\beta_{\text{fwd}}(q))). \tag{4.7}$$

Equations (4.4), (4.5), (4.6), and (4.7) 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 4.2.

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

$$\delta_{\text{fwd}}(o) = \max(\max_{k\in\{1,\ldots,n\}}(f(h(o),k)+p(h(o),k)), \max_{(q,k)\in Q}(f(h(q),k)+p(q,k)+\delta_{\text{fwd}}(q))). \tag{4.8}$$

Equations (4.1), (4.2), (4.3), and (4.8) apply to computing memory costs whether a graph representation represents multidimensional array variables and vectorized operations, or scalar variables and nonvectorized operations.

4.4 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 3.2.2.

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 3.2.2 is at least

$$\sum_{i}^{n} \text{size}\left(v_i\right), \tag{4.9}$$

where n is the number of nodes in the graph representation and $v_i \in \mathscr{V}$ 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

$$\sum_{i}\sum_{j} \text{size}\left(\frac{\partial b_i}{\partial a_j}\right), \tag{4.10}$$

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 (4.10) to be an overestimate of the number of nodes in the computational graph. In other cases, (4.10) 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.

4.5 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 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 $G'(\mathscr{N}', \mathscr{E}')$ be a DAG, where the edge set $\mathscr{E}' = \{(u,w) : (u,v),(v,w) \in \mathscr{E}, v \in \mathscr{O}\}$ represents dependencies between variables ($u,w \in \mathscr{V}$ for each $(u,w) \in \mathscr{E}$) and $\mathscr{N}' = \{v : v \in \mathscr{V}\}$ represents variables. Let $t : \mathscr{E} \to \mathbb{R}_+$ map each edge to a value representing the computation time of an operation

$v \in \mathscr{O}$. 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 $e \in \mathscr{E}'$ corresponds to an operation $v \in \mathscr{J}$, where $\mathscr{J} \subseteq \mathscr{O}$ 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 $e \in \mathscr{E}'$ corresponds to an operation $v \in \mathscr{I}$, where $\mathscr{I} \subseteq \mathscr{O}$ is the set of implicit operations, then the time t for the operation represented by $e \in \mathscr{E}'$ to execute is as high as $t(e) = T(h(v))m_v$, where h is described at the end of Chapter 4, $m_v \in \mathbb{N}$ is the maximum number of iterations used to solve the implicit function represented by v, and

$$T(o) = \sum_{i=1}^{k} t(x_i) \tag{4.11}$$

is the total time to evaluate all operations in series within a model contained within the operation $x \in \mathscr{O}$. To compute an estimate of the execution time exploiting all available opportunities for parallelization, only the $x_i \in \mathscr{O}$ on the critical path need to be included in Equation (4.11).

4.6 Run-Time Costs

This section presents the results of estimating the run-time costs of various system models based on the analyses presented in Chapter 4. The time and memory costs for nine engineering models are estimated based on their graph representations of Section 3.2.2. Figures 4.1, 4.2, 4.4, and 4.5 are based on nine engineering models defined using CSDL (Section 3.4): a model of a propeller rotor using BEM theory [20], a model of a propeller rotor using the blade element momentum-based ideal-loading design (BILD) method [66], a model of airflow over an electrical vertical takeoff and landing (eVTOL) vehicle concept using the vortex lattice method (VLM) [9], an equivalent circuit model (ECM) of a battery [9], 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) [67], a model used for general aviation aircraft

weight estimation ("Weights") [9], and a model of wing, rotor, fuselage, and tail geometries with mesh generation ("Geometry") [9].

Figure 4.1 shows the relative time cost of evaluating a computational model when exploiting all opportunities for parallelization[1] for each of the nine models presented in this section. The compiler can compute these estimates of the parallelized execution time by applying Equation (4.11) to the graph representation of a model. Compiler back 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.

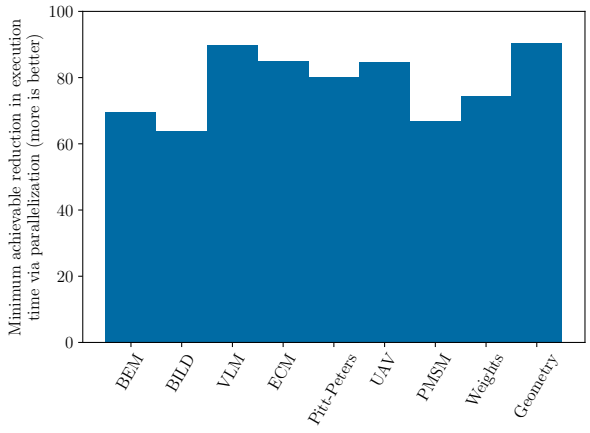


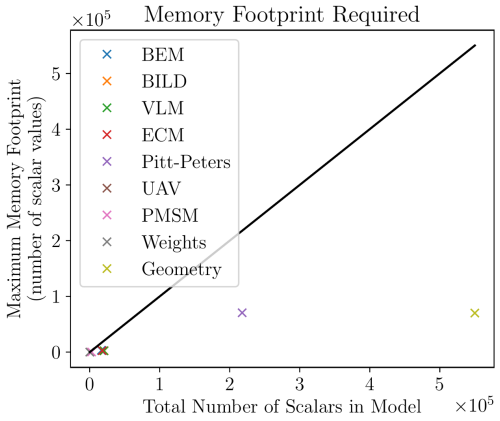
Figure 4.1. The graph representation for each model is used to predict the time to evaluate a computational model using all opportunities for parallelization as a percentage of the time to evaluate a computational model without exploiting any opportunities for parallelization. Note that all operations are assumed to take the same amount of time to execute.

Figures 4.2, 4.4, and 4.5 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 4.2 shows the total number of scalar values used to define

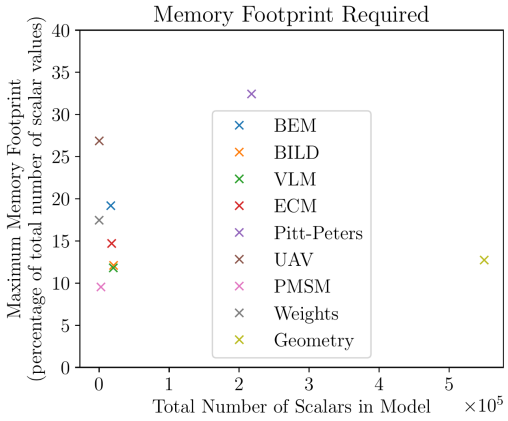
[1] Execution of individual vectorized operations may be further parallelized, reducing computation time by more than what is shown in Figure 4.1.

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 4.2 and 4.4 is computed by applying Equation (4.3) to the graph representation for each of the nine engineering models under consideration.

Figure 4.2 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.



(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

Figure 4.2. Maximum number of scalar values to allocate at any point during program execution.

Figure 4.4 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 4.3 shows visualizations and results from three low-fidelity models using a compiler front end that generates a graph representation from code written in CSDL and a compiler back end that generates OpenMDAO code. Figure 4.3 also shows the adjacency matrices of the graph representation for each system-level model.

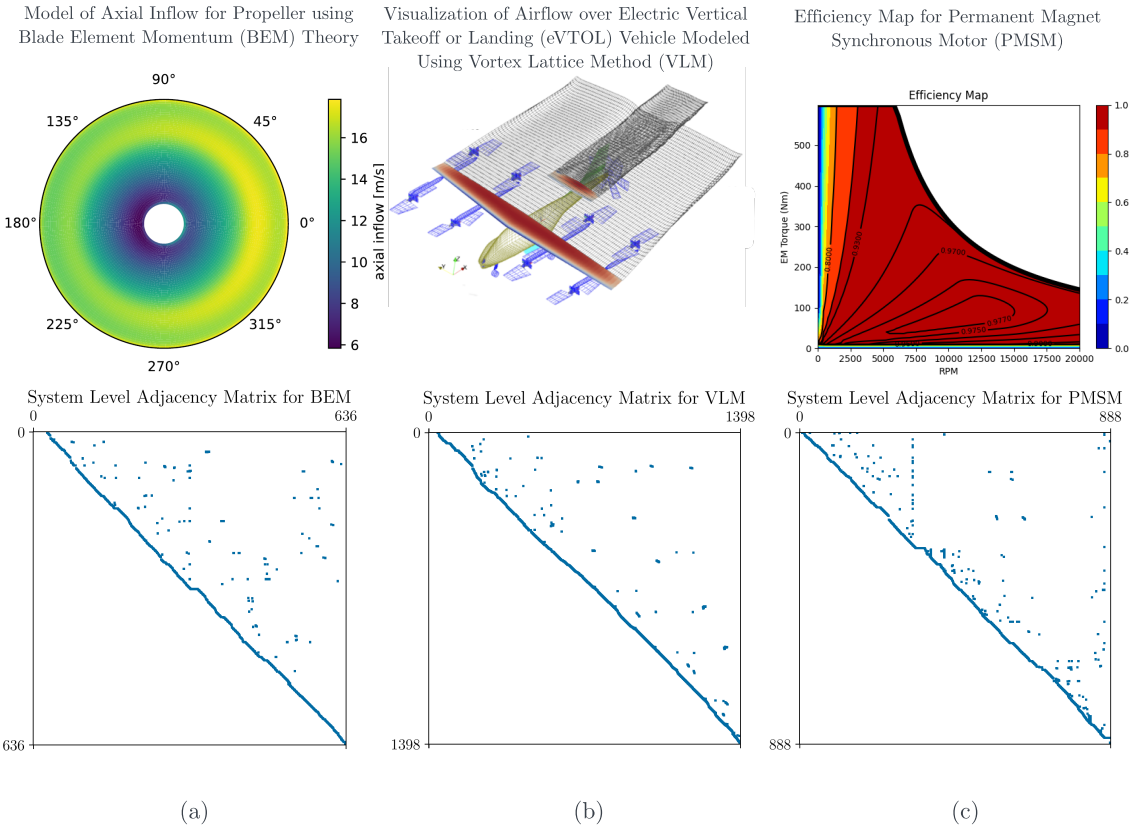


Figure 4.3. Three models generated using CSDL and the system-level adjacency matrices corresponding to their compiler graph representations: (a) axial inflow velocity for a propeller modeled using blade element momentum (BEM) theory [66], (b) airflow over an electrical vertical takeoff and landing (eVTOL) vehicle concept using the vortex lattice method (VLM) [9], and (c) an efficiency map for a permanent magnet synchronous motor (PMSM) [67]. The computational models corresponding to the images in the top row were compiled using a compiler back end that constructs an OpenMDAO model.

Figure 4.5 shows an estimate of the maximum number of scalar values to allocate at

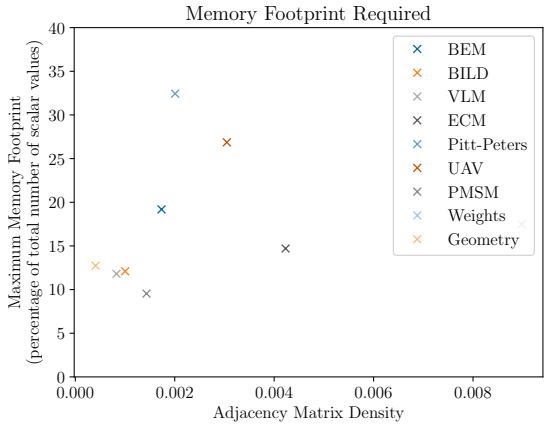


Figure 4.4. Percentage of the total number of scalar values used to define the model that are allocated during program execution versus the density of the adjacency matrix of the model's graph representation. The density remains below 1% for all models examined, and there is a small positive correlation between overall memory footprint and the density of the adjacency matrix.

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 4.5 indicate opportunities for dynamically allocating chunks of memory temporarily during model evaluation.

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 forward or reverse-mode differentiation. Figure 4.6 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

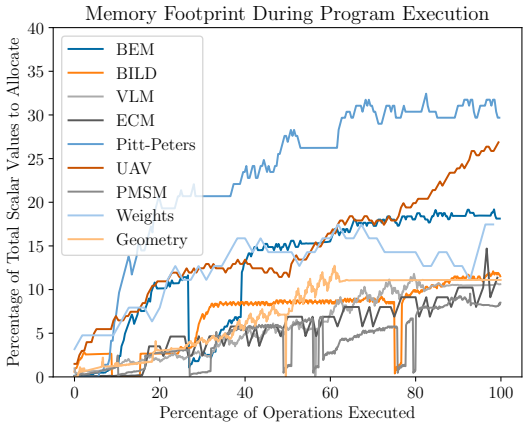


Figure 4.5. number of scalar values that must be allocated at each vectorized operation during model evaluation. The number of values that must be allocated rises to the number of values used to define model outputs (e.g., objective and constraints). These values must remain in memory until the model is evaluated to provide external access to the resulting values.

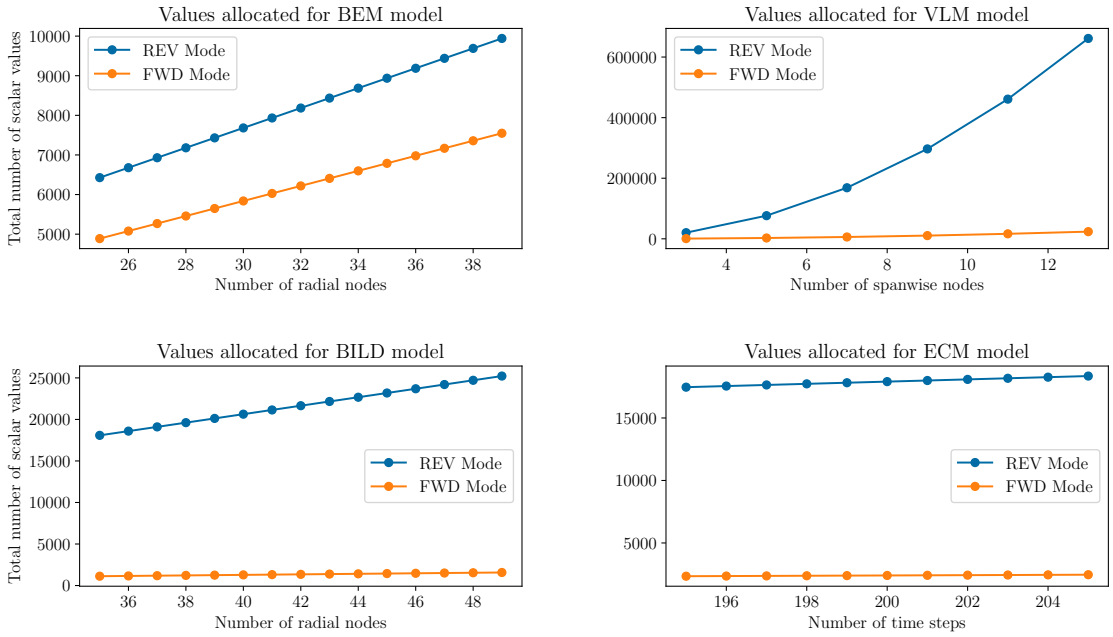


Figure 4.6. Required memory at run time computed by applying Equations (4.1) and (4.3) to the graph representation (Section 3.2.2) for various models defined in CSDL, as a function of a key parameter. Clockwise from top left: a model of a propeller rotor using BEM theory; an aerodynamic model of an eVTOL vehicle concept using the VLM; an ECM of a battery; and a model of a propeller rotor using BILD method.

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 4.6 shows how increasing the number of spanwise nodes increases the memory requirement quickly when solving an optimization problem containing a 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 4.6 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.

4.7 Conclusion

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.

4.8 Acknowledgments

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

Chapter 4, in full, has been submitted for publication of the material as it may appear in Structural and Multidisciplinary Optimization 2023. Gandarillas, Victor; Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT, Springer Nature, 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Large-scale Multidisciplinary Design Optimization of a Virtual-Telescope CubeSat Swarm

5.1 Introduction

CubeSats and SmallSats are classes of miniature satellites that are becoming increasingly popular for scientific and commercial missions due to their low cost and versatility. They have been used for a variety of missions, including Earth observation and scientific research. They also offer a relatively affordable alternative to traditional large-scale missions and allow for increased frequency of launches. In recent years, SmallSats and CubeSats have seen a significant growth in mission diversity, including Earth observation, planetary exploration, space technology development, and scientific research.

A diversity of mission requirements has led to a diversity of CubeSat and SmallSat designs. CubeSats such as the Radiometer Assessment using Vertically Aligned Nanotubes (RAVAN) [68], designed to measure the Earth's outgoing radiation, and the Lunar Flashlight [69], designed to search for ice on the Moon's surface, also showcase the diversity of satellite designs. The SmallSat X-ray Quantum Calorimeter Satellite (XQCSat) [70, 71] is designed to perform X-ray spectroscopy.

One advantage of CubeSats and SmallSats is their small size, which makes them easy

to launch and allows for multiple spacecraft to be launched at once. Some satellite missions require multiple spacecraft to form a distributed platform and fly in formation. The TerraSAR-X and Tandem-X (TerraSAR-X add-on for Digital Elevation Measurement) [72] spacecraft use Synthetic Aperture Radar (SAR) to produce high-resolution images of the Earth's surface and generate a Digital Elevation Model (DEM) of the Earth's surface. The TechSat-21 [73] was a technology demonstration mission focused on demonstrating new technologies for formation flight. TanDEM-X and TechSat-21 are both remote sensing systems, but SmallSats have filled the need for formation flight beyond Earth orbit as well. The Double Asteroid Redirection Test (DART) [74, 75] was a technology demonstrator aimed at redirecting near-Earth objects (NEOs) that successfully redirected an asteroid.

Virtual telescopes represent another application of CubeSats and SmallSats, which leverage formation flying to observe celestial objects. These platforms offer a cost-effective alternative to traditional telescopes, which can be expensive to build and maintain. Virtual telescopes are formed from SmallSats, such as the Miniature Distributed Occulter Telescope (mDOT) [76], Virtual Telescope for X-ray Observations (VTXO) [77, 78, 78, 79], and Proba-3 [80, 81] missions, or CubeSats, such as the VIrtual Super-resolution Optics with Reconfigurable Swarms (VISORS) [82, 83, 84].

Multidisciplinary design optimization enjoys a long history of solving spacecraft design problems successfully [85, 86, 87, 22]. MDO is the use of numerical optimization to generate designs for engineering systems that involve multiple subsystems or disciplines [88]. Early studies explored various optimization techniques for applying MDO to spacecraft design [85], including the Solar, Anomalous, and Magnetospheric Particle Explorer (SAMPEX) mission where modified dynamic programming algorithm was found to guarantee an optimum, but suffered from exponential time complexity [89].

Software frameworks dedicated to spacecraft MDO problems have also been developed based on the collaborative optimization (CO) method [86, 87]. The CO method is a multilevel concurrent optimization approach that solves an optimization problem at the discipline level

before solving an optimization problem at the system level. At the discipline level, the constraints for each disciplines are satisfied without taking into acount interactions with other disciplines. The system level optimization problem is solved to resolve the inconsistencies across disciplines and the find the compatibility optimal design [86]. CO has the disadvantage of slow convergence and often does not converge [86]. Some coupled variables are even excluded from the system-level optimization to reduce complexity at the system level [87].

Gradient-based optimization techniques have emerged as a promising technique for large-scale MDO problems [22, 4, 88, 7, 90]. Gradient-based methods can be applied to MDO problems with objective and constraint functions that are continuous and differentiable in the design variables, also referred to as nonlinear programs (NLPs) and scale better with the number of design variables than gradient-free approaches. A disadvantage of gradient-based approaches is that they require derivative information from the model of the system, which can be difficult to implement for large-scale multidisciplinary systems, especially when there is coupling across disciplines. Computing exact derivatives for models with multiple disciplines often relies on computational architectures [4], software frameworks [7], and modeling languages [90] that facilitate multidisciplinary model definition, evaluation, and derivative computation. The large-scale MDO of a CubeSat mission has been demonstrated previously with the CubeSat investigating Atmospheric Density Response to Extreme driving (CADRE), a remote-sensing Earth observation mission [22]. The CADRE MDO algorithm involved several disciplines: communication, battery, thermal, electrical, attitude dynamics, and orbit dynamics. These disciplines are all tightly coupled, and due to the simulation of 9 orbits, the NLP contains tens of thousands of variables.

This chapter is concerned with a mission inspired by the VISORS mission. VISORS is a CubeSat swarm mission with the objective of observing active solar regions with a virtual telescope, where the optics and detector reside on separate spacecraft [82, 83, 84]. As such, VISORS imposes tight alignment and separation constraints during the observation phase of the mission, which couples the dynamics of all the CubeSats in the swarm. Major challenges

present themselves with a virtual telescope that are not present with a single-CubeSat mission. First, tight alignment and pointing constraints are imposed on the swarm. These constraints are on the order of tens of millimeters while the positions of the spacecraft relative to the center of the Earth are on the order of thousands of kilometers, requiring an approach that captures multi-scale effects without introducing numerical error. Prior approaches to virtual telescope design have focused on designing one spacecraft subsystem using gradient-free approaches. For example, the attitude control for VTXO is optimized in [77] using neural networks and a genetic algorithm, a gradient-free approach.

This chapter applies gradient-based MDO to the design of a virtual telescope mission with two CubeSats flying in formation. The problem definition uses a relative orbit model that captures orbit perturbations and whose numerical accuracy is robust to the size of the orbit of the spacecraft and the number of spacecraft in a swarm. Solving the complete design problem simultaneously was found not to converge robustly, but convergence to an optimal solution was robust when using an approach that solves a series of optimization problems sequentially. The successful design of a virtual telescope mission demonstrates MDO as a practical, scalable approach to solving spacecraft mission design problems involving swarms flying in formation.

The chapter is organized as follows. Section 5.2 describes the approach used to apply MDO in this study. Section 5.3 presents an overview of the mission design. Section 5.4 presents an overview of the spacecraft discipline models. Section 5.5 presents the optimization problems solved using the approach in Section 5.2. Section 5.6 presents results for this study. Section 5.7 concludes the chapter.

5.2 Approach

This section presents the approach for solving the problem of designing a virtual telescope using multidisciplinary design optimization (MDO). A virtual telescope is a telescope where the main components such as the optics and detector reside on different spacecraft (Figures 5.2, 5.3).

Designing a virtual telescope is a large-scale MDO problem with complex interactions across spacecraft and spacecraft subsystems. The interactions between the spacecraft models occur through the separation and pointing constraints during the observation phase of the mission. The telescope formation places constraints on the interactions between the spacecraft, and power requirements lead to constraints on variables within the individual spacecraft subsystems.

The large number of variables and complex interactions between the spacecraft disciplines and the spacecraft subsystem disciplines present a challenge to finding an optimal design for a virtual-telescope mission. Therefore, an approach that handles a large number of variables and complex interactions between disciplines is necessary to achieve an optimal design for a virtual-telescope mission.

The approach used in this chapter is a gradient-based approach that solves the trajectory, attitude profile, and battery pack optimization problem. Gradient-based approaches are preferable to gradient-free approaches when the problem objective and constraints are continuous and differentiable functions of the design variables because they converge to a solution much more quickly than gradient-free methods. Since gradient-free methods need to explore the entire design space, the number of function evaluations required for convergence grows as an exponential function of the number of the design variables. Alternatively, gradient-based methods exhibit a quadratic rate of convergence near a local optimum. It is important to note that neither gradient-free nor gradient-based methods guarantee convergence to a global optimum.

This mission, however, presents some obstacles to using gradient-based methods. First, gradient-based methods can only be applied to problems where the objective and constraint functions are continuous and differentiable in the design variables. Second, computing derivatives for all the functions in the system model scales poorly in terms of development time and effort unless a framework provides some level of automation for computing derivatives. Third, some discipline models can be discrete functions, which prevent the use of gradient-based methods unless a well-justified continuous and differentiable approximation to the discrete model can be incorporated into the system model.

The first obstacle is that not all design variables in the optimization problem are continuous quantities, and the problem must be modified in order to use a gradient-based approach. Four design variables are integer design variables: the number of battery cells connected in each series, and the number of battery cell series connected in parallel, for each of the two spacecraft (Table 5.3). With the introduction of integer design variables, the problem becomes a mixed-integer nonlinear program (MINLP). To solve the MINLP, the problem is relaxed so that the number of battery cells connected in each series and the number of battery cell series connected in parallel are real numbers. If the solution is not an integral solution, a branch-and-bound method is used.

The second challenge of implementing derivative computation for large-scale models is overcome by using the adjoint method to compute derivatives. The cost of adjoint-based derivative computation is largely independent of the number of inputs (design variables) when the number of inputs is greater than the number of outputs (objective and constraints) [26]. Although the cost of the adjoint method does not scale with the number of design variables, which leads to slower convergence when solving the optimization problem. Constraints on array variables, e.g., variables that are the result of discretized functions of time, may be aggregated into one constraint to take full advantage of faster derivative computation offered by the adjoint method. Constraints are aggregated into a single inequality constraint using the Kresselmeier–Steinhausser (KS) function [91]. The KS function is a continuous and differentiable function that aggregates constraints across all time steps into one constraint by approximating the maximum value of an array. The KS function is given by

$$\mathrm{KS}(g(x)) = g_{max}(x) + \frac{1}{\rho}\ln\sum_i \exp[\rho(g_i(x) - g_{max}(x))], \tag{5.1}$$

where $g_i(x)$ is the i^{th} value of the function $g(x)$, $g_{max}(x)$ is the maximum value of the function $g(x)$, and ρ is a smoothing parameter. As ρ approaches infinity, the value of the KS function approaches the true maximum. However, a large value of ρ leads to poor conditioning of the

derivatives, so a moderate value of ρ must be selected. The value of ρ is problem dependent. By enforcing an upper (lower) bound on the maximum (minimum) value of an array, the KS function encourages the optimizer to resolve the largest infeasibilities first. In this chapter, the KS function (5.1) is modified such that $\mathrm{KS}(kg(x))/k$ is used with $\rho = 1$, where k is a large number. In this application, using the KS function in this way appears to be behave well with arbitrarily large k.

The implementation overhead of applying the adjoint method is also a factor in the decision to use the adjoint method to compute derivatives. Fortunately, software frameworks such as OpenMDAO [7] and CSDL [90] are available to facilitate implementation of derivatives for large-scale MDO, making gradient-based optimization accessible to users. In this chapter, the MDO problem is defined and solved using CSDL. The key advantages of using CSDL over other frameworks are that: (1) CSDL provides a natural way to compose models and their interactions, and (2) CSDL fully automates adjoint-based derivative computation for large-scale, coupled systems. CSDL is used to define the optimization problem and the reduced-Hessian SQP solver SNOPT [92] is used to solve the optimization problems summarized in Section 5.5.

The third challenge of applying gradient-based methods to optimization problems arises when physics-based models are not available and system modelers only have access to discrete data points for a given system (e.g., discrete functions, experimental data). This challenge is overcome by training continuous and differentiable surrogate models on discontinuous data. All surrogate models in this chapter rely on regularized minimal-energy tensor-product B-splines (RMTB) [93]. RMTB models provide fast predictions for low-dimensional models and do not suffer from numerical issues when trained using large data sets.

5.3 Mission Overview

This section provides an overview of the mission to which the approach described in Section 5.2 is applied. Section 5.4 presents overviews of each discipline involved in the

model. The mission under consideration is inspired by the Virtual Super-resolution Optics with Reconfigurable Swarms (VISORS), a space-based virtual telescope whose primary mission objective is to capture high-resolution images of active solar regions to enable testing fundamental theories of coronal heating on the Sun [82]. Greater observational capability generally requires larger telescopes capable of collecting more light and/or focusing at greater distances.

A space-based virtual telescope is a compelling concept for space-based telescope design that is comprised of more than one spacecraft, each carrying different elements of the telescope. Because a virtual telescope is not restricted to house the components on a single spacecraft, the mass of the telescope remains relatively constant as the dimensions of the telescope increase.

The dimensions of the telescope depend more on the in-flight formation than on the size of the individual spacecraft. To ensure that launch costs scale well with the size of the telescope, the trajectory must be chosen to minimize the total propellant used.

Each orbit contains an observation and standby phase (Figure 5.1). In the observation phase, the telescope makes scientific observations of the corona. In order to form a telescope, the spacecraft must satisfy tight alignment, separation, and pointing constraints during the observation phase, obtaining accurate measurements of the target area on the corona. In the standby phase, the spacecraft must satisfy power requirements. The virtual telescope in this study is comprised of two CubeSats, one carrying the optics portion of the telescope, and the other carrying the detector used to record images. Each CubeSat is referred to by the scientific equipment it carries: the spacecraft carrying the optics is called the optics spacecraft (OSC), and the spacecraft carrying the detector is called the detector spacecraft (DSC).

Each observation requires that the OSC and DSC maintain telescope formation for one 10 s interval per orbit. During the observation phase, the spacecraft are required to maintain a nominal separation of 40 m$\pm$15 mm, alignment within 18 mm, and pointing error below 90 arcsec (Figure 5.2).

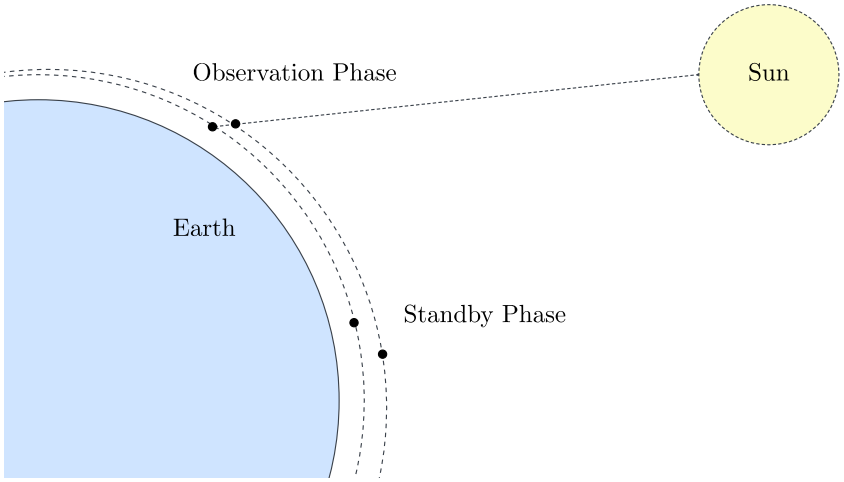


Figure 5.1. Concept of operations for the virtual telescope based on the Virtual Super-resolution Optics with Reconfigurable Swarms (VISORS) mission.

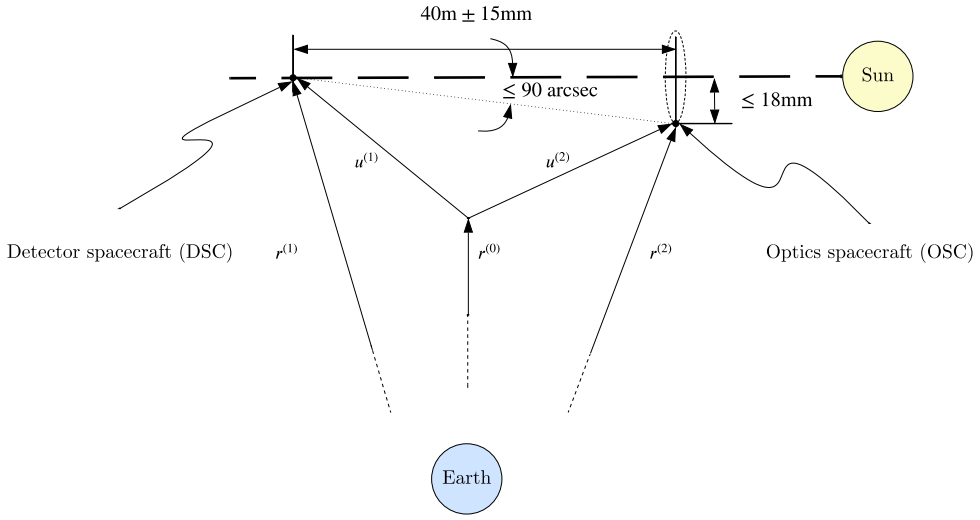


Figure 5.2. Spacecraft with positions $r^{(i)}$ relative to Earth, and positions $u^{(i)}$ relative to reference orbit $r^{(0)}$ with alignment, separation, and pointing requirements.

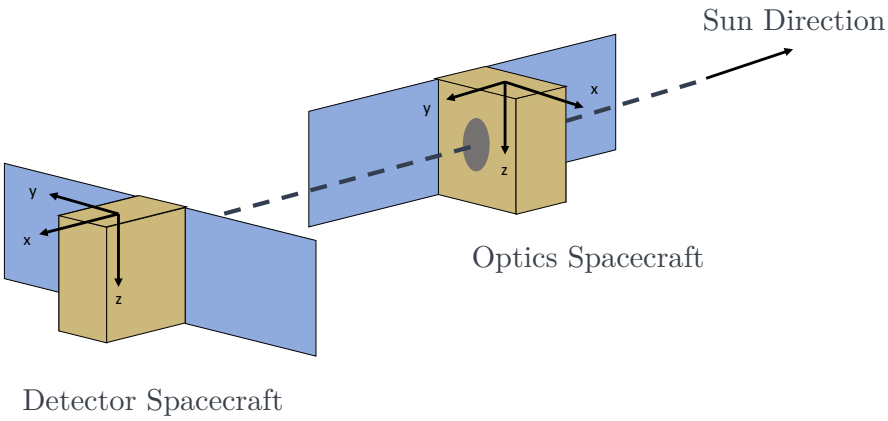


Figure 5.3. CubeSats shown in their attitude configuration during observations of the Sun. The body reference frame of each spacecraft is shown fixed in the body of each spacecraft. The detector spacecraft is oriented such that light from the Sun enters the detector and the optics spacecraft is oriented such that light from the Sun passes through the optics onboard the optics spacecraft. Solar arrays are shown in blue.

5.4 Discipline Models

5.4.1 Propulsion

Each CubeSat is equipped with thrusters to control the translational motion. The propulsion discipline for each CubeSat is modeled such that each CubeSat has omnidirectional translational control. Omnidirectional translational control requires six thrusters firing in the direction of the positive and negative directions along each axis of the body frame. For simplicity, the thrust over time $F_{\text{thrust},i}(t), i \in \{1,2,3\}$ is defined in the Earth-centered Inertial (ECI) frame, which is the same frame used in the orbit dynamics model (Section 5.4.2), and can be translated to the body frame to model the amount of force that each thruster imparts to the spacecraft.

The thrusters used for this mission are cold-gas thrusters in accordance with the VISORS concept of operations [82]. The initial propellant mass m_0 for each CubeSat is a design variable in the optimization problem. The propellant mass evolves over time according to the differential equation

$$\dot{m}(t) = -\frac{\sum_{i=1}^{3} |F_{\text{thrust},i}(t)|}{g_0 I_{\text{sp}}}, \tag{5.2}$$

where g_0 is the acceleration due to gravity at sea level and I_{sp} is the specific impulse of the cold gas thruster. The absolute value is used to capture the fact that propellant is always used regardless of the direction of thrust. Since the absolute value function is not differentiable at zero, the number of thrust design variables is doubled: one set of design variables represents the positive values of each component of thrust over time, and the other set represents the negative values of each component of thrust over time. For gradient-based optimization, however, the absolute value function is discontinuous at zero, so the thrust design variables are split into a set of positive and negative values.

5.4.2 Orbit Dynamics

This section presents a relative orbit model that can be used for optimizing trajectories of spacecraft swarms subject to millimeter-level constraints. The new orbit model can be used within an MDO framework to design swarm missions with any number of spacecraft.

The key challenge of trajectory optimization with millimeter-level constraints lies in computing the positions of the spacecraft with the numerical precision necessary for solver convergence. In low earth orbit (LEO), the equations of motion for the spacecraft orbits are functions of the positions of spacecraft relative to the Earth, which are on the order of thousands of kilometers, while the constraints on the dimensions of the swarm formation are on the order of tens of millimeters. This is a difference of eight orders of magnitude.

When integrating the equations of motion of the spacecraft positions relative to the center of the Earth, large differences in orders of magnitude introduce truncation errors, rendering numerical optimization impossible. As a result, the optimizing solver cannot update the constraint values with the required level of precision to converge to a feasible solution.

A model of the orbits of the spacecraft in the swarm that does not introduce truncation errors is developed to make numerical optimization of the swarm design problem possible. To reduce the relative orders of magnitude between the constraint values, design variable values, and intermediate state variable values, a trajectory for each spacecraft relative to a reference orbit is used. The position of the i^{th} spacecraft $r^{(i)}$ (Figure 5.2) is given by

$$r^{(i)} = r^{(0)} + u^{(i)}, \tag{5.3}$$

where $r^{(0)} \in \mathbb{R}^3$ is a reference orbit position, and $u^{(i)} \in \mathbb{R}^3$ is the position of the i^{th} spacecraft relative to the reference orbit. The equations of motion for the spacecraft relative to the reference orbit $r^{(0)}$ are given by

$$\ddot{u}^{(i)} = \ddot{r}^{(i)} - \ddot{r}^{(0)}. \tag{5.4}$$

The orbit dynamics of the reference orbit $r^{(0)}$ and the spacecraft positions relative to the center of the Earth $r^{(i)}$ are modeled by

$$\begin{aligned}
\ddot{r} = -\frac{\mu}{r^3} r &- \frac{3\mu J_2 R_e^2}{2r^5}\left[\left(1 - \frac{5r_z^2}{r^2}\right) r + 2r_z\hat{z}\right] \\
&- \frac{5\mu J_3 R_e^3}{2r^7}\left[\left(3r_z - \frac{7r_z^3}{r^2}\right) r \right. \\
&\qquad \left. + \left(3r_z - \frac{3r^2}{5r_z}\right) r_z\hat{z}\right] \\
&+ \frac{15\mu J_4 R_e^4}{8r^7}\left[\left(1 - \frac{14r_z^2}{r^2} + \frac{21r_z^4}{r^4}\right) r \right. \\
&\qquad \left. + \left(4 - \frac{28r_z^2}{3r^2}\right) r_z\hat{z}\right].
\end{aligned} \tag{5.5}$$

The J_2, J_3, and J_4 terms capture perturbations that affect the orbit on the order of months, whereas the orbital period for LEO is generally on the order of 90 minutes. The use of MDO will capture these effects across multiple time scales, as well as the effect of rotating the orbital plane on

power generation. Note that if $\ddot{r}^{(i)}$ and $\ddot{r}^{(0)}$ are modeled using Equation (5.5), then $\ddot{r}^{(0)}$ is a function of $r^{(0)}$, and $\ddot{r}^{(i)}$ is a function of $r^{(i)} = u^{(i)} + r^{(0)}$. Expanding Equation (5.4) by modeling $\ddot{r}^{(0)}$ and $\ddot{r}^{(i)}$ using Equation (5.5) leads to

$$\begin{aligned}
\ddot{u} = &-\frac{3\mu J_2 R_e^2}{2\|r^{(0)}\|^5}\left[\left(1-\frac{5(r_z^{(0)}+u_z)^2}{\|r^{(0)}\|^2}\right)u+2u_z\hat{z}\right] \\
&-\frac{5\mu J_3 R_e^3}{2\|r^{(0)}\|^7}\left[\left(3(r_z^{(0)}+u_z)-\frac{7(r_z^{(0)}+u_z)^3}{\|r^{(0)}\|^2}\right)u\right. \\
&\qquad\left.+\left(6r_z^{(0)}+3u_z\right)u_z\hat{z}\right] \\
&+\frac{15\mu J_4 R_e^4}{8\|r^{(0)}\|^7}\left[\left(1-\frac{14(r_z^{(0)}+u_z)^2}{\|r^{(0)}\|^2}+\frac{21(r_z^{(0)}+u_z)^4}{\|r^{(0)}\|^4}\right)u\right. \\
&\qquad\left.+\left(4-\frac{28}{3\|r^{(0)}\|^2}(3(r_z^{(0)})^2+3r_z^{(0)}u_z+u_z^2)\right)u_z\hat{z}\right].
\end{aligned} \tag{5.6}$$

Then the $\ddot{u}^{(i)}$ in Equations (5.4) and (5.6) are functions of $r^{(0)}$, and $u^{(i)}$ with no explicit dependence on $r^{(i)}$. Since $r^{(0)}$ does not depend on any design variables, the magnitude of $r^{(0)}$ has no impact on the precision of the constraints imposed on the $u^{(i)}$ or their derivatives. Furthermore, the reference orbit $r^{(0)}$ can be computed once prior to optimization, reducing computation time each time the model is evaluated. Figure 5.4 shows two trajectories of free-flying spacecraft relative to a reference orbit. These trajectories are computed by integrating Equation (5.6) for each spacecraft using initial conditions corresponding to the desired orbit in Table 5.1.

The desired orbit parameters in Table 5.1 are based on the orbit requirements for the VISORS mission [83]. The mission requirements call for a telescope maintaining a sun-synchronous orbit due to the ease of accessibility by secondary payloads and the slow and predictable change in beta angle throughout the year [84].

5.4.3 Attitude Dynamics and Control

Both the observation phase and standby phase of the orbit impose requirements on the spacecraft behavior that determines the attitude profiles of the spacecraft. The pointing

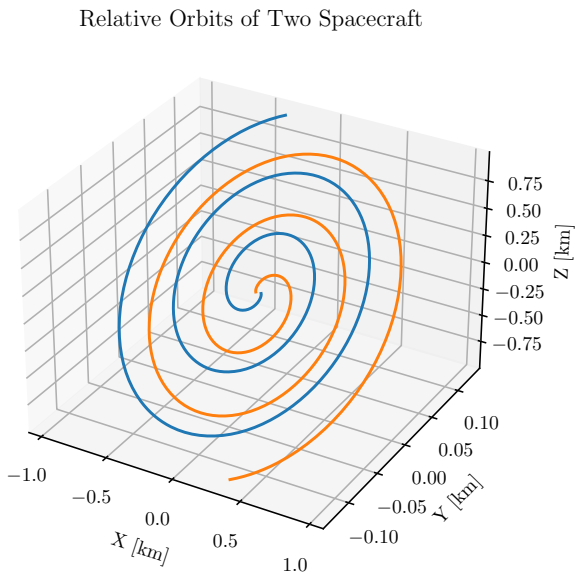


Figure 5.4. Positions of two free-flying spacecraft relative to a sun-synchronous reference orbit over 3 orbits around the Earth. The initial conditions of the spacecraft are offset from the initial condition for the reference orbit by 20 m in the $+x$ and $-x$ directions. Coordinates are in the ECI frame of reference.

Table 5.1. VISORS orbit requirements, preferred orbit, and orbit-insertion tolerance

Orbital Parameter	Requirement	Preferred	Tolerance
Altitude	450-600km	500km	$\pm$ 50km
Inclination	$\geq 30^\circ$	97.4°	$\pm 0.2^\circ$
Eccentricity	≤ 0.1	0.01	± 0.001

requirements imposed on the spacecraft to maintain the telescope formation dictate the attitude profile during the observation phase. Power requirements imposed on the electrical power subsystem to maximize mission lifetime constrain the attitude profile throughout the standby phase. In this chapter, the Euler angles (roll, pitch, and yaw) defining the attitude of each spacecraft are optimized to satisfy the individual spacecraft pointing and sizing constraints. During the observation phase of the orbit, the spacecraft must point its telescope components toward the Sun to record observations, and the battery state of charge must remain within 20-95% throughout the orbit to minimize battery degradation while maintaining a power balance. The

battery and solar power models are presented in Sections 5.4.6 and 5.4.5, respectively. The reaction wheels controlling the attitude of the spacecraft are also limited to a maximum torque and power of 0.004 Nm and 1 W, respectively, to match the Blue Canyon Technologies RWP015 reaction wheel specifications [94]. The reaction wheel constraints, listed in Table 5.3, are enforced using a KS function (5.1) in Section 5.2.

First, a rotation matrix is constructed to transform coordinates from the ECI frame to the spacecraft body frame by

$$R_{B/I} = \begin{bmatrix} c_\theta c_\psi & c_\theta s_\psi & -s_\theta \\ s_\phi s_\theta c_\psi - c_\phi s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\theta s_\phi \\ c_\phi s_\theta c_\psi + s_\phi s_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi & c_\theta c_\phi \end{bmatrix},$$

where $s_{(\cdot)}$ and $c_{(\cdot)}$ denote the sine and cosine of an angle, respectively, and ψ, θ, ϕ are roll, pitch, and yaw, respectively. The rotational dynamics of the spacecraft (in the body frame) are given by

$$\dot{L}_B = \tau_B + \tau_{RW} + \omega_B \times (L_{RW}) + \tau_g = 0,$$

where L_B is the angular momentum of the spacecraft, $\tau_{RW} = J_{RW} \cdot \dot{\omega}_{RW}$ is the torque imparted to the spacecraft by reaction wheels, and $\tau_g$is the torque imparted to the spacecraft by the Earth's gravity. The torques τ_B, τ_g are given by,

$$\tau_B = J_B \cdot \dot{\omega}_B + \omega_B \times (J_B \cdot \omega_B) + \tau_g, \tag{5.7}$$

$$\tau_{g,x} = -3(J_{B,y} - J_{B,z})\Omega^2 (R_{B/A})_{21}(R_{B/A})_{31}, \tag{5.8}$$

$$\tau_{g,y} = -3(J_{B,z} - J_{B,x})\Omega^2 (R_{B/A})_{31}(R_{B/A})_{11}, \tag{5.9}$$

$$\tau_{g,z} = -3(J_{B,x} - J_{B,y})\Omega^2 (R_{B/A})_{11}(R_{B/A})_{21}, \tag{5.10}$$

where Ω is the osculating angular speed of the spacecraft position in the orbit and $R_{B/A}$ is the rotation matrix from the orbit frame to the body frame given by

$$R_{B/A} = \begin{bmatrix} \hat{r}^T \\ ((\hat{r} \times \hat{v}) \times \hat{r})^T \\ (\hat{r} \times \hat{v})^T \end{bmatrix}.$$

Using a constant orbit angular speed models the attitude dynamics under the assumption that the orbit is circular, while using the osculating orbit angular speed takes into account the effect of orbit perturbations.

The angular velocity of the spacecraft ω_B is known from computing $\omega_B^{\times} = \dot{R}_{B/I}(R_{B/I})^T$, where $\dot{R}_{B/I}$ is computed via finite-differences. The angular acceleration of the spacecraft $\dot{\omega}_B$ is also computed via finite-differences. The resulting ordinary differential equation

$$\dot{\omega}_{RW} = J_{RW}^{-1}(\omega_B \times (J_{RW} \cdot \omega_{RW}) - \tau_B)$$

is then integrated to obtain reaction wheel angular velocities $\omega_{RW}(t)$. Once the reaction wheel angular velocities are computed, the reaction wheel torques τ_{RW} and power draw for each reaction wheel $¶_{RW}$ are computed by

$$\tau_{RW} = J_{RW}\dot{\omega}_B \tag{5.11}$$

$$P_{RW} = \tau_{RW}\omega_B. \tag{5.12}$$

5.4.4 Solar Illumination

The amount of power generated by the solar arrays depends on the amount of solar illumination. The solar illumination is a function of the orientation of the spacecraft relative to the Sun, the placement of the solar arrays on the spacecraft, and the dimensions of the solar arrays. To compute the solar illumination, a ray-tracing algorithm is provided by the Toolbox for Analysis and Large-scale Optimization of Spacecraft (TALOS) [1]. Figure 5.5 shows training and test data for the solar arrays used in this study.

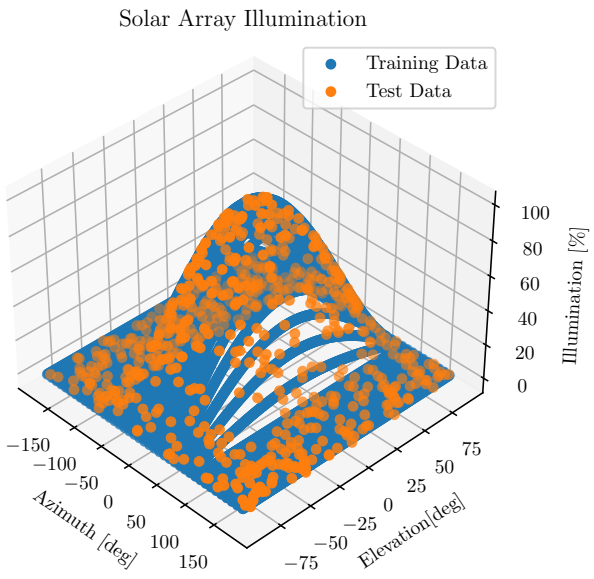


Figure 5.5. Solar illumination as a percent of the area of solar arrays illuminated as a function of azimuth and elevation. The training data is generated using a ray-tracing algorithm provided by the Toolbox for Analysis and Large-scale Optimization of Spacecraft [1].

The percent solar illumination depends on the position of the CubeSats in the orbit as well as its attitude. For each CubeSat, a line-of-sight indicator is also computed as shown in Figure 5.6. The Sun line-of-sight indicator is defined as

$$LOS_s = \begin{cases} 1 & \hat{r}_{b/e} \cdot \hat{r}_{s/e} \geq 0 \\ \begin{cases} 1, & d_s > R_e \\ 3\eta^2 - 2\eta^2, & \alpha R_e < d_s < R_e \\ 0, & d_s < \alpha R_e \end{cases} & \hat{r}_{b/e} \cdot \hat{r}_{s/e} < 0 \end{cases},$$

where

$$d_s = \|\hat{r}_{b/e} \times \hat{r}_{s/e}\|_2 \quad \eta = \frac{d_s - \alpha R_e}{R_e - \alpha R_e}.$$

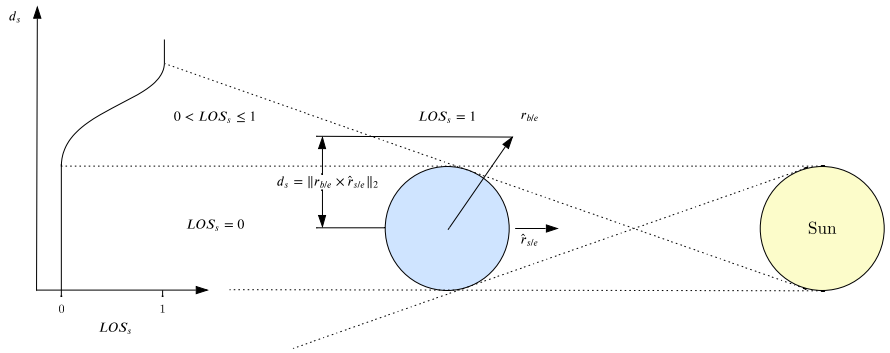


Figure 5.6. Schematic of Sun line-of-sight (LOS_s) variable indicating when each spacecraft has a line of sight to the Sun. When $LOS_s = 0$, the spacecraft is hidden behind the Earth's shadow, and when $LOS_s = 1$, the spacecraft has a line of sight to the Sun. A gradient-based approach to optimization cannot handle the discontinuous jump between the two states, so a continuous and differentiable function is used. The continuous and differentiable function is also physically justified by modeling the umbra and penumbra effects of the Earth's shadow.

5.4.5 Solar Power

The solar power discipline models the power supplied by the solar arrays on the spacecraft. For this study, the solar power discipline described in [95] is used to model the current and voltage of the solar arrays. The solar array current I is a design variable in the optimization problem. The solar array voltage V is given by

$$V(I) = \frac{V_{OC} + V_0}{2} + \frac{V_{OC} - V_0}{2} \tanh\left(b(I - I_{sc}) + \operatorname{arctanh}\left(\frac{V_0 + V_{OC}}{V_0 - V_{OC}}\right)\right), \tag{5.13}$$

where

$$\begin{aligned} b &= \frac{1}{V_{OC} - V_0}\frac{dV}{dI}, \\ \frac{dV}{dI} &= -\frac{V_T}{V_T + I_{sat}Rsh}R_{sh}, \\ V_{OC} &= 1.1V_T \ln\frac{V_T}{I_{sat}R_1}, \\ R_1 &= 1 \text{ (by assumption)}, \end{aligned} \tag{5.14}$$

and the constants and variables in 5.13 are listed in [23]. The solar array discipline in [95] is an explicit approximation of the model described in [96]. Figure 5.7 shows the I-V curve for the solar arrays.

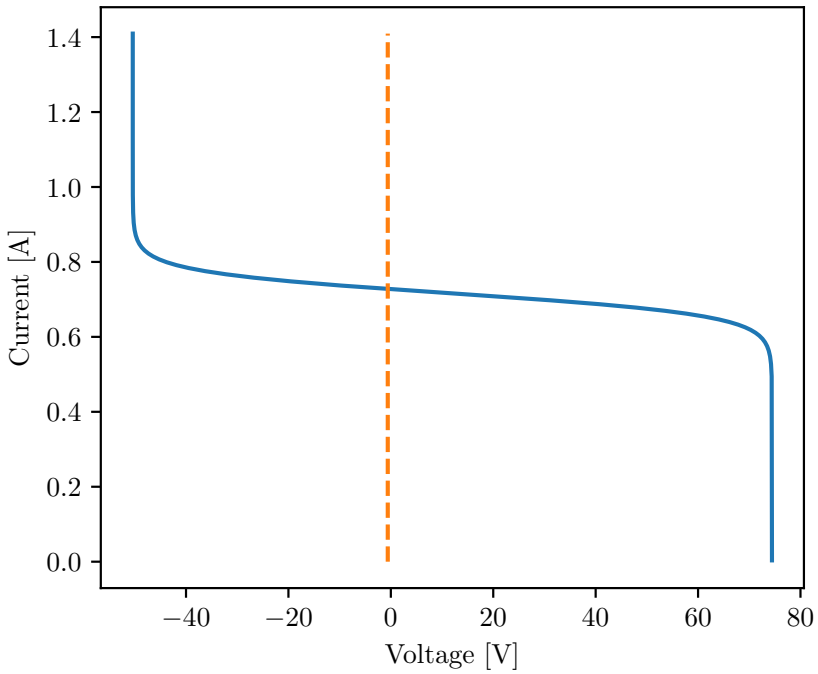


Figure 5.7. I-V curve for solar arrays.

5.4.6 Battery Model

Each CubeSat needs to be able to store enough power to deliver to other subsystems for the mission. The spacecraft subsystems drain power from the battery pack, limiting its use. Each CubeSat is equipped with solar arrays to recharge its batteries using energy from the Sun in flight. Using rechargeable batteries greatly extends the life of the mission. Charging and discharging the batteries too much can lead to premature degradation, however. To maximize the lifetime of the CubeSats, the state of charge of the batteries is constrained to be within 20-95%. The attitude profile (Section 5.4.3) must be designed to satisfy this constraint so that the solar arrays are oriented toward the Sun during the standby phase long enough to store the energy required to

power the spacecraft subsystems, but not so long that the batteries are overcharged. Each battery cell is modeled by a Thevenin equivalent circuit model shown in Figure 5.8.

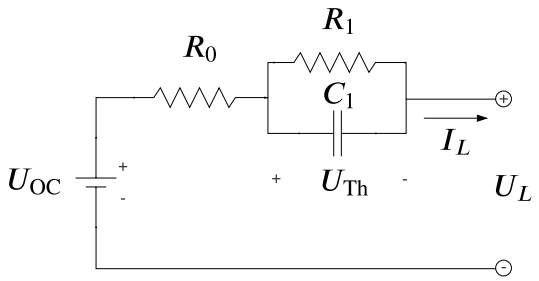


Figure 5.8. Equivalent circuit model of a battery cell using a Thevenin model with a single RC element.

The battery cell state of charge z and voltage U_L are modeled by

$$\begin{aligned} \dot{z}(t) &= \frac{I_L}{Q}, \\ U_L &= U_{\mathrm{OC}} - U_{\mathrm{Th}} - I_L R_0, \end{aligned} \tag{5.15}$$

where z is the state of charge (SOC), Q is the cell capacity, I_L is the current, U_{OC} is the open-circuit voltage of the cell, R_0 is the internal resistance of the cell, and U_L is the voltage across the battery cell. Note that the battery cell itself is not an RC circuit, but the behavior of the cell is modeled by an equivalent circuit. The Thevenin voltage U_{Th} is modeled by the differential equation

$$\dot{U}_{\mathrm{Th}} = -\frac{U_{\mathrm{Th}}}{R_1 C_1} + \frac{I_L}{C_1}, \tag{5.16}$$

where R_1 is the polarization resistance in the RC circuit and C_1 is the equivalent capacitance in the RC circuit. The state of charge z is limited to be between 20% and 95%.

The values for the equivalent circuit model are computed using

$$U_{\mathrm{OC}}, C_1, R_1 R_0 = f(z), \tag{5.17}$$

where f is a surrogate model generated from data for a lithium-ion battery provided by [97].

The battery pack current, voltage, and power are given by

$$\begin{aligned} I_{\text{bat}} &= I_L n_p, \\ U_{\text{bat}} &= U_L n_s, \\ P_{\text{bat}} &= I_{\text{bat}} U_{\text{bat}}, \end{aligned} \tag{5.18}$$

where n_s and n_p are the number of battery cells connected in each series and number of battery cell series connected in parallel, respectively. The total number of battery cells in the battery pack is given by $n_s n_p$. The variables n_s and n_p are design variables in the optimization problem given by Table 5.3.

5.5 Optimization

This section describes the optimization problem solved using the approach described in Section 5.2 to design the mission presented in Section 5.3. The problem of maximizing its mission lifetime is solved by solving two optimization problems: a trajectory optimization problem summarized in Table 5.2, and then an attitude profile and battery pack optimization problem summarized in Table 5.3. The problem summarized in Table 5.3 is solved for each spacecraft. Solving these problems sequentially is justified by the fact that the solution to the trajectory optimization problem is not affected greatly by the attitude and electrical power subsystem disciplines.[1] Figures 5.9 and 5.10 contain a variant of the eXtended Design Structure Matrix (XDSM) diagram [98] of the multidisciplinary systems in the problems summarized in Tables 5.2 and 5.3, respectively. These diagrams serve to show the data flow across disciplines. For the sake of brevity, Figures 5.9 and 5.10 show the major disciplines of the system models to emphasize the general structure of the models used to define each optimization problem.

[1] As a general rule, this is not the case. For example, if instead of omnidirectional thrust, the spacecraft were equipped with unidirectional thrusters, then the attitude discipline could not be decoupled from the trajectory optimization problem.

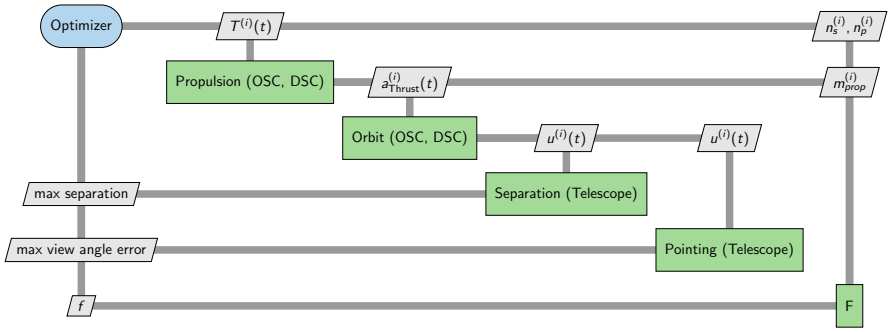


Figure 5.9. XDSM diagram for problem summarized in Table 5.2.

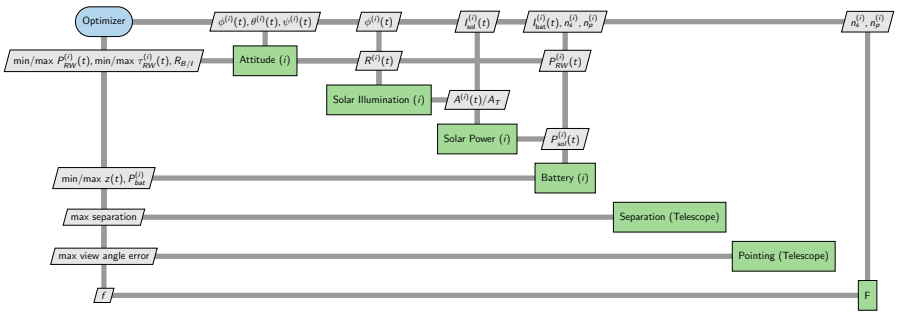


Figure 5.10. XDSM diagram for problem summarized in Table 5.3.

The procedure for maximizing mission lifetime is as follows. First, the trajectory optimization problem (Table 5.2) is solved over one orbit with the constraints shown in Figure 5.2 imposed on the relative positions of the two spacecraft. The objective of the first problem is to minimize the total propellant used over one orbit. Constraints are enforced on the separation between the two spacecraft and the pointing error of the telescope. The alignment constraint shown in Figure 5.2 is redundant and not included in this problem definition. Then, the orbit from the trajectory optimization problem is fed into the problem summarized in Table 5.3 that simultaneously optimizes the attitude profile and battery model. The objective of the second problem is to minimize the number of battery cells in the battery pack. Constraints are enforced on the reaction wheels and battery pack according to Table 5.3.

There are two scalar design variables per spacecraft; the number of battery cells connected in each series, and the number of battery cell series connected in parallel. Since these variables are integer design variables, the problem is a mixed-integer nonlinear program (MINLP).

The model in all optimization problems simulates a single orbit, approximately 90 minutes, split into 301 time steps. The profile variables were discretized with $n_t = 301$ points and the time-dependent design variables were represented using fourth-order B-splines with $n_{cp} = 60$ control points.

Table 5.2. Trajectory Optimization Problem

	Variable	**Description**	**Quantity**
Objective	$\sum_{i=1}^{2}(m_{\text{prop},0} - m_{\text{prop,f}})$	Total propellant mass used	
Design Variables	$-T_{max} \leq T(t) \leq 0$	Thrust in negative direction of each component (omnidirectional)	$2 \times 3 \times n_{\text{cp}}$ (a B-spline interpolation is used to obtain thrust for all time)
	$0 \leq T(t) \leq T_{max}$	Thrust in positive direction of each component (omnidirectional)	$2 \times 3 \times n_{\text{cp}}$ (a B-spline interpolation is used to obtain thrust for all time)
		Total no. of design variables	$12n_{\text{cp}}$
Constraints	$450 - h(t) \leq 0$	Altitude	2 (reduced from $2n_t$ using KS function)
	$(\|u^{(DSC)}(t) - u^{(\text{OSC})}(t)\| - 40 - \varepsilon)LOS_s \leq 0$	Separation (along LOS)	1 (reduced from n_t using KS function)
	$(40 - \varepsilon - \|u^{(DSC)} - u^{(\text{OSC})}\|)LOS_s \leq 0$	Separation (along LOS)	1 (reduced from n_t using KS function)

Continued on next page

Table 5.2 Trajectory Optimization Problem – continued from previous page

	Variable	Description	Quantity
	$\|(u^{(\mathrm{OSC})}(t) - u^{(DSC)}(t)) \cdot \hat{r}_{sun}\| LOS_s - \varepsilon \leq 0$	Pointing	1 (reduced from n_t using KS function)
		Total no. of constraints	5

Table 5.3. Attitude & Battery Optimization Problem

	Variable	Description	Quantity
Objective	$n_s n_p$	Total number of battery cells	
Design Variables	$0 \leq I \leq I_{sc,0}$	Solar power current	1
	$0.2 \leq z_0 \leq 0.95$	Initial state of charge	1
	$n_s \geq 1, n_p \geq 1$	Number of battery cells connected in series and in parallel	2
	$\phi_{B/I}(t)$, $\theta_{B/I}(t)$, $\psi_{B/I}(t)$	Spacecraft roll, pitch, and yaw in ECI frame	$3 \times n_{\mathrm{cp}}$ (a B-spline interpolation is used to obtain thrust for all time)
		Total no. of design variables	$4 + 3n_{\mathrm{cp}}$
Constraints	$0.2 - z(t) \leq 0$	State of charge	1 (reduced from n_t using KS function)
			Continued on next page

Table 5.3 Attitude & Battery Optimization Problem – continued from previous page

Variable	Description	Quantity
$z(t) - 0.95 \leq 0$	State of charge	1 (reduced from n_t using KS function)
$-\tau_{RW,max} - \tau_{RW}(t) \leq 0$	Reaction wheel torque along each axis	3 (reduced from $3 \times n_t$ using KS function)
$\tau_{RW}(t) - \tau_{RW,max} \leq 0$	Reaction wheel torque along each axis	3 (reduced from $3 \times n_t$ using KS function)
$I_{batt}(t) - 5 \leq 0$	Battery charge rate	1 (reduced from n_t using KS function)
$-10 - I_{batt}(t) \leq 0$	Battery discharge rate	1 (reduced from $1n_t$ using KS function)
$(1 - \varepsilon - (R_{B/I}^{(\mathrm{OSC})}(t)\hat{r}_{sun})_y)LOS_s \leq 0$	optics spacecraft orientation	1 (reduced from n_t using KS function)
$(1 - \varepsilon - (R_{B/I}^{(\mathrm{DSC})}(t)\hat{r}_{sun})_x)LOS_s \leq 0$	detector spacecraft orientation	1 (reduced from n_t using KS function)
	Total no. of constraints	11

5.6 Results

The optimization problems summarized in Tables 5.2 and 5.3 were defined using CSDL [90], and sovled using SNOPT [92], a reduced-Hessian active-set SQP solver to solve the optimization problems. I used the modOpt[2] framework as an interface between CSDL

[2]https://lsdolab.github.io/modopt/

and SNOPT. Section 5.6.1 presents the solution to the problem summarized in Table 5.2 and Section 5.6.2 presents the solution to the problem summarized in Table 5.3. All inequality constraints are aggregated using a KS function (5.1) such that $\mathrm{KS}(kg(x))/k$ is used with $\rho = 1$, where k is a large number.

5.6.1 Trajectory Optimization

The overall objective is to maximize the mission lifetime. The problem is split into a trajectory optimization problem and for each spacecraft, an attitude profile and battery pack optimization problem. The objective of the trajectory optimization problem is to minimize the total propellant used. The trajectory optimization converges to a feasibility of 10^{-5} and an optimality of 10^{-6} (Figure 5.11). The convergence history in Figure 5.11 shows that feasibility converges quickly relative to optimality, indicating that once the formation requirements are satisfied, they remain satisfied until the propellant mass is minimized. Figure 5.12 shows the positions of the optics and detector spacecraft (OSC and DSC, respectively) relative to the reference orbit after solving the problem summarized in Table 5.2. The positions of the spacecraft relative to the reference orbit at the solution are on the order of hundreds of meters. Figure 5.12 shows that the reformulated orbital dynamics are capable of reducing the magnitude of the intermediate state variables (in this case, the positions of the spacecraft relative to a reference orbit) to a level low enough to avoid the introduction of truncation errors that would otherwise prevent optimization.

Figure 5.13 shows the values of the constraint variables on the telescope formation over time. The constraints are only enforced during the observation phase (highlighted in green). Figure 5.13 shows the successful optimization of the trajectory to satisfy the millimeter- and arcsecond-level constraints on the swarm formation. The spacecraft break formation soon after the end of the observation phase, suggesting that the formation must be maintained actively.

Figure 5.14 shows the optimal thrust profile for each spacecraft. The thrust for both spacecraft varies the most prior to and during the observation phase and drops to zero for the rest

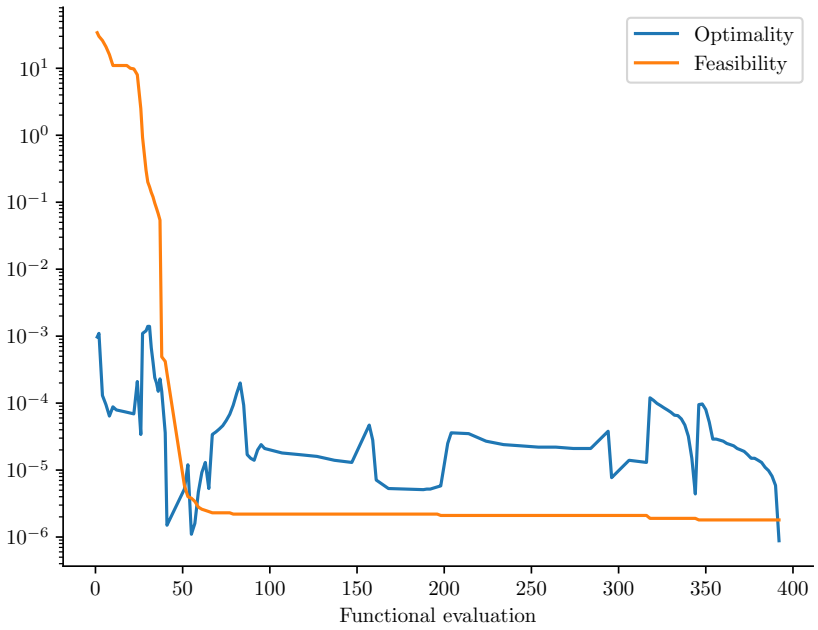


Figure 5.11. SNOPT convergence history for problem summarized in Table 5.2.

of the orbit. The thrust is nonzero shortly after the observation phase. This is due to the use of a B-spline interpolation to reduce the number of design variables. The thrust profile is discretized into 301 time steps and the thrust design variables are 60 B-spline interpolants. Each interval between the B-spline interpolants lasts for 95 s, which is about the length of time that the thrust is nonzero after the observation phase.

Figure 5.15 shows the history of the values of the objective and aggregated constraint values for the trajectory optimization problem. The constraint values are aggregated from time-dependent constraints using the KS function (5.1). As mentioned in the discussion surrounding Figure 5.11, the constraint values approach the values representing the formation requirements early in the optimization and the total propellant used is minimized thereafter. The total propellant used starts at zero because the thrust design variables are initialized to zero prior to the first function evaluation. The total propellant used is approximately equal for each spacecraft, suggesting that the final design use the same or similarly sized propellant tanks for both spacecraft.

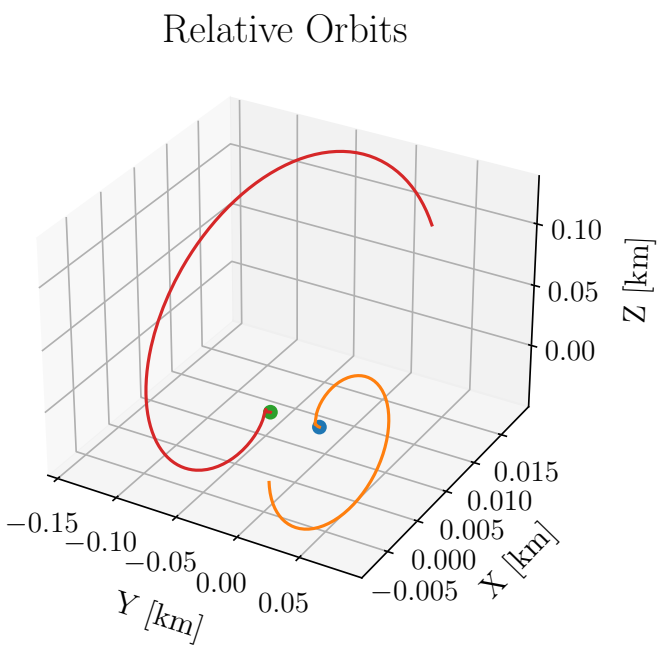


Figure 5.12. 3D plot of relative orbits of both spacecraft relative to reference orbit at optimal solution of problem summarized in Table 5.2. Coordinates are in the ECI frame of reference.

The constraint on maximum separation over the entire orbit becomes inactive early in the optimization. This suggests that if there are any destabilizing effects on the separation between the spacecraft, they are not strong enough to affect the solution over one orbit.

5.6.2 Attitude Profile and Battery Pack Optimization

This section presents the solution to the problem summarized in Table 5.3 for each spacecraft. The orbit from the solution presented in Section 5.6.1 is fed into in the problem summarized in Table 5.3. The problem is solved independently for each spacecraft. The attitude and battery pack optimizations converge to a feasibility of at most 10^{-4} and an optimality of at most 10^{-4} (Figure 5.16) for both the OSC and DSC. The rates of convergence for both spacecraft are different due to the initial guess for the attitude profile of each spacecraft so that the telescope components are oriented toward the Sun as shown in Figure 5.3.

Figure 5.17 shows the number of battery cells connected in each series, number of

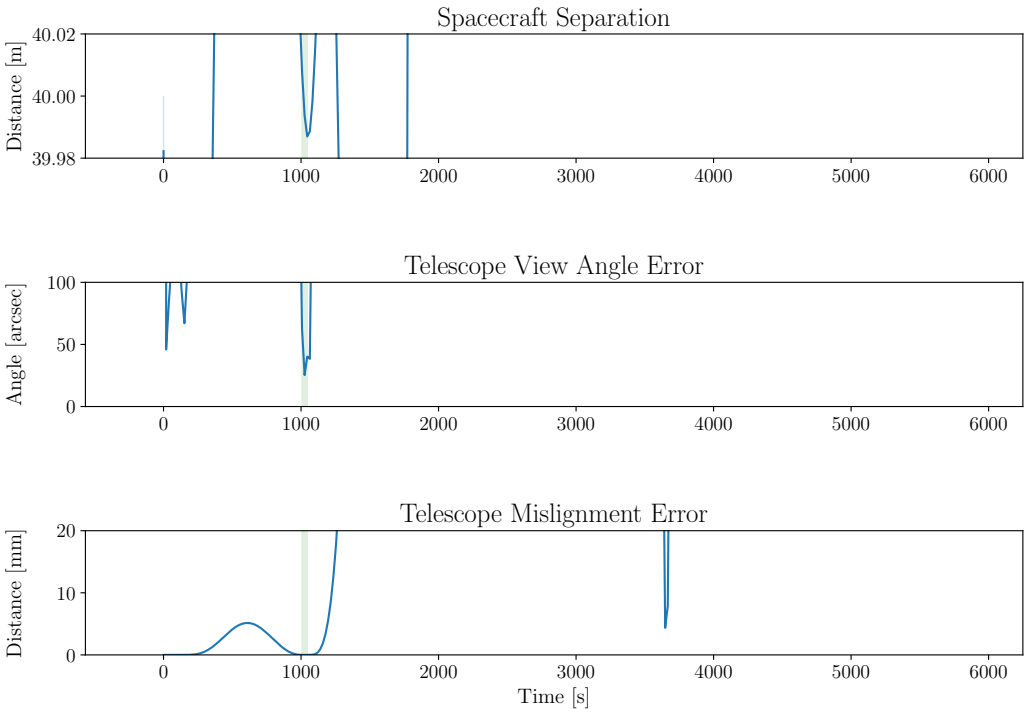


Figure 5.13. Telescope dimensions over one orbit. The green highlighted vertical strip indicates the observation phase of the orbit. The formation constraints are satisfied during the observation phase (separation is 40 m$\pm$15 mm, telescope pointing error is $\leq$90 arcsec, and telescope misalignment is $\leq$18 mm). The formation is maintained throughout the observation phase, which lasts for 38 s, exceeding the 10 s requirement.

battery cell series connected in parallel, and maximum telescope component exposure during the observation phase for the OSC and DSC at the solution to the problem summarized in Table 5.3. The number of battery cells connected in each series n_p and the number of battery cell series connected in parallel n_p are each initialized to 2 prior to optimization and drop to their lower bound of 1 early in the optimization. The total number of battery cells in the later iterations and at the solution is $n_s n_p = 1$, indicating that the battery cell capacity is large enough to power the subsystems for this configuration. Since the design variables n_s and n_p are both 1 at the solution, they are both integer values, which yield a solution to the mixed-integer nonlinear program (MINLP) without the need to use a branch-and-bound approach. The minimum exposure for each spacecraft reaches 98% early in the optimization as well.

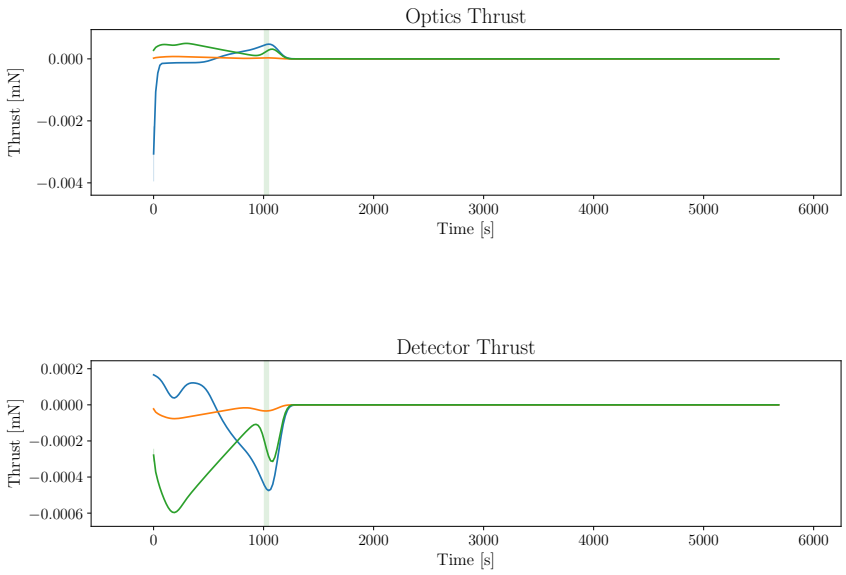


Figure 5.14. Thrust profile for each spacecraft over one orbit. Thrusters have omnidirectional capability, and each component of thrust is in the ECI frame of reference. The green highlighted vertical strip indicates the observation phase of the orbit.

Figure 5.18 shows the roll, pitch, and yaw angles of the OSC and DSC at the solution, and the transition between the observation phase and standby phase of the orbit. The transition between observation phase and standby phase is more apparent in the OSC attitude profile because both spacecraft have the same initial guess for their respective attitude profiles, but different desired orientations during the observation phase (Figure 5.3, Table 5.3).

Figure 5.19 shows the percent exposure, a measure of the amount of light from the Sun that reaches the telescope component throughout the orbit. During the observation phase, both spacecraft achieve an exposure of 98% or greater. The detector on the DSC is oriented towards the Sun for more of the orbit than the optics on the OSC is oriented toward the Sun. This is due to the fact that the optics and the detector are oriented differently relative to the solar arrays on each spacecraft, requiring a different orientation during the observation phase for each spacecraft (Figure 5.3).

Figure 5.20 shows the state of the battery pack onboard each spacecraft at the optimal

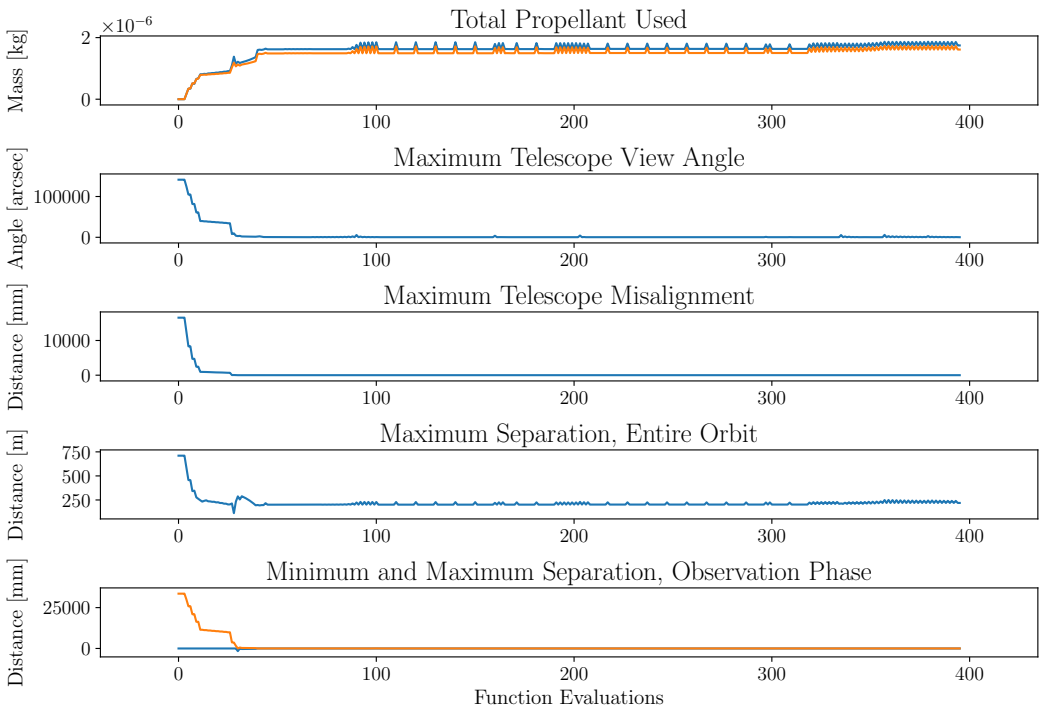
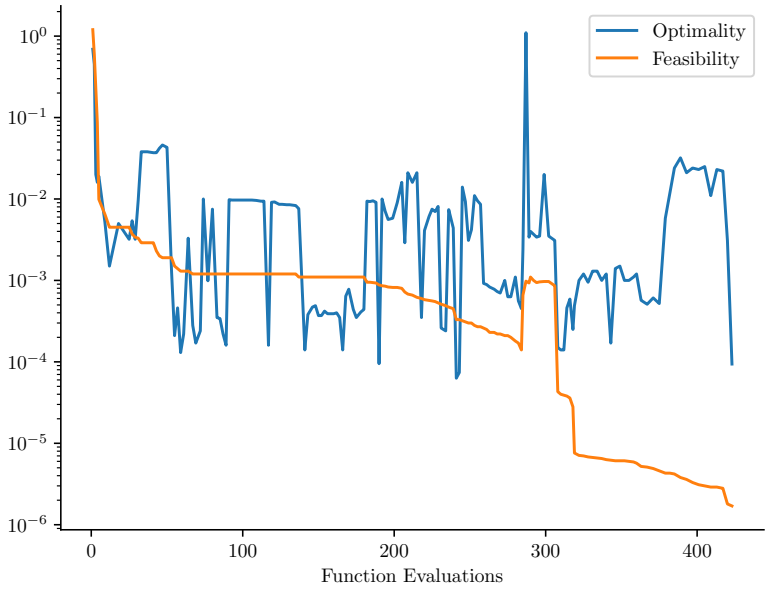


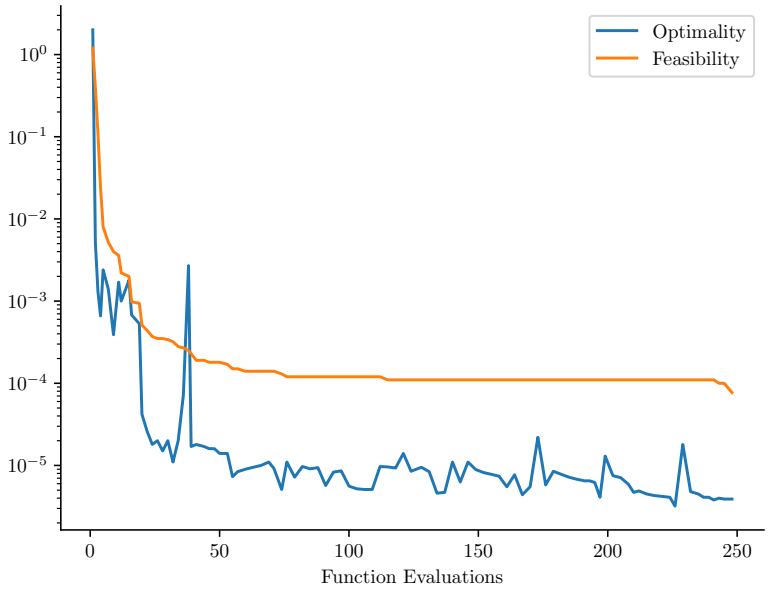
Figure 5.15. Objective and constraint history for trajectory optimization problem summarized in Table 5.2

solution. Both battery states of charge remain well above 20% and reach a maximum of 95%. The state of charge for both batteries decreases prior to the observation phase as both spacecraft reaction wheels consume power to achieve the desired orientation, and both spacecraft orient their solar arrays to drive the state of charge back to its value at the start of the orbit. The battery current, voltage, and power vary much less over time for the OSC than for the DSC, indicating that battery packs onboard the two spacecraft do not have the same demands placed on them.

Figure 5.21 shows the reaction wheel speed, torque, and power over time at the solution. The reaction wheel speeds, the torque applied by the reaction wheels, and the power supplied to the reaction wheels do not have behaviors that show the observation phase and standby phase clearly. This indicates that the torque applied by the Earth's gravity field is a significant torque for the reaction wheels to counteract throughout the orbit.

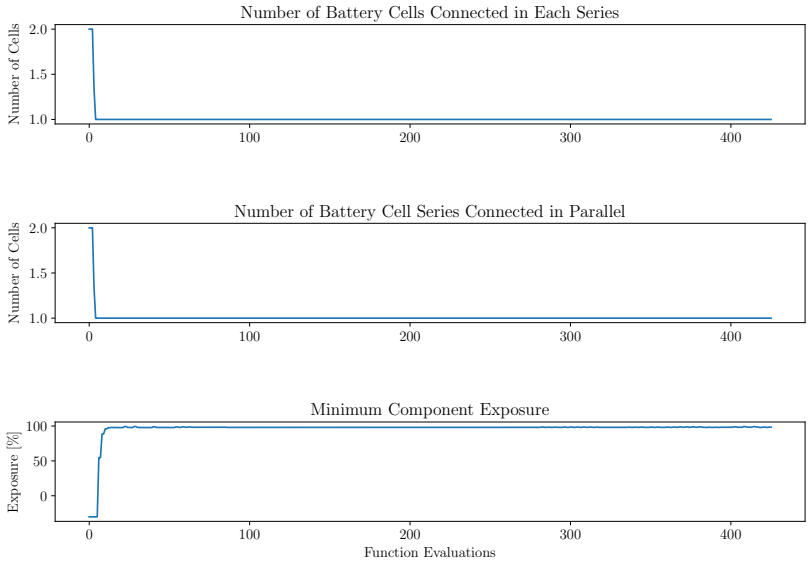


(a)

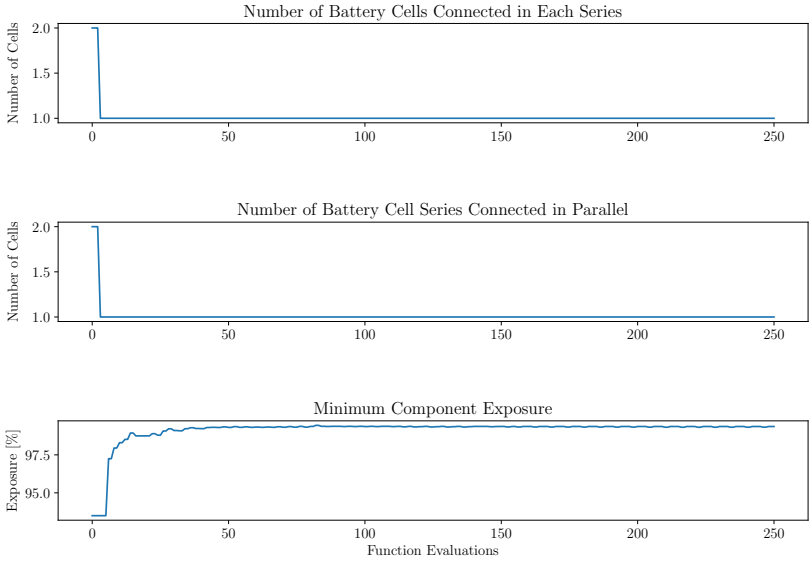


(b)

Figure 5.16. SNOPT convergence history for problem summarized in Table 5.3 applied to (a) optics spacecraft and (b) detector spacecraft.



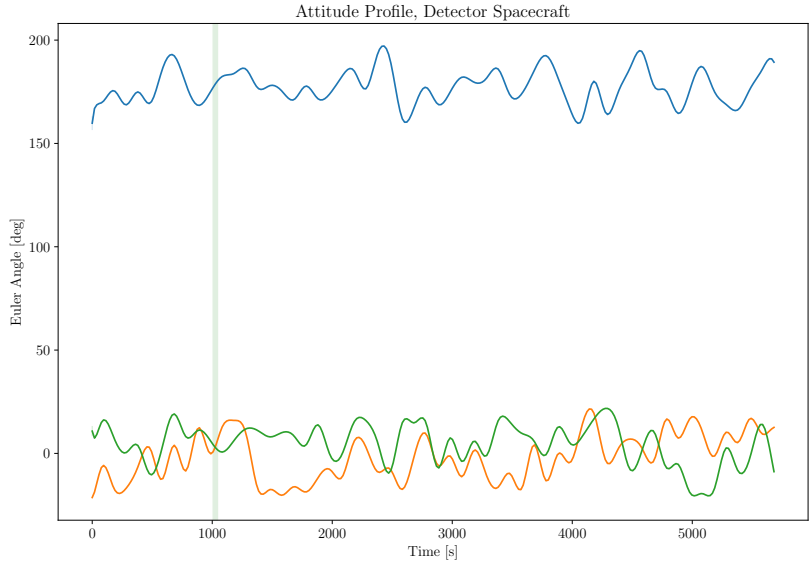
(a)



(b)

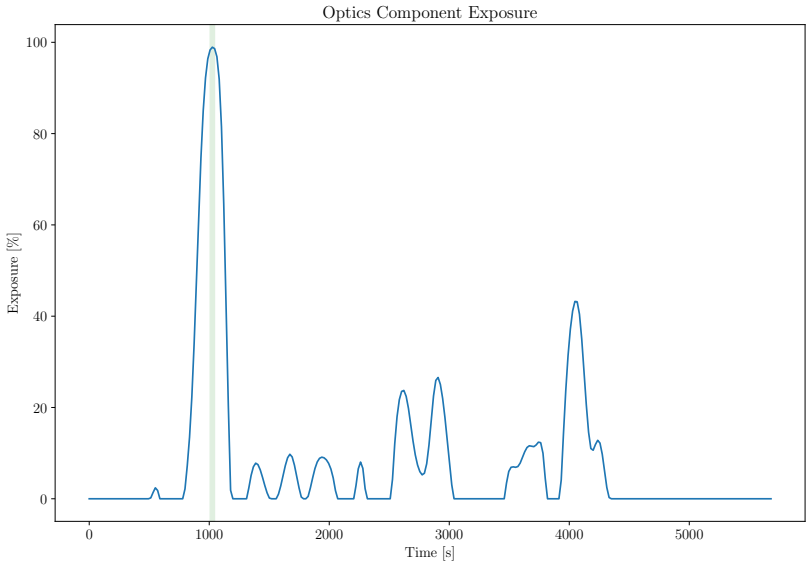
Figure 5.17. Scalar design variables and constraints for problem summarized in Table 5.3. Each plot shows the history of the design variable values for the battery pack sizing (number of battery cells connected in each series and number of battery cell series connected in parallel) and the exposure of each telescope component on the (a) optics spacecraft and (b) detector spacecraft.

(a)

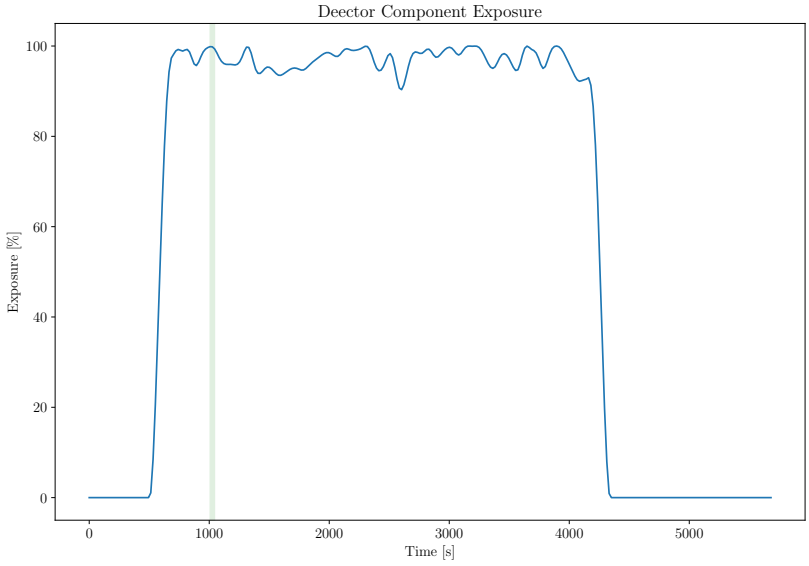


(b)

Figure 5.18. Attitude profile of (a) optics and (b) detector spacecraft as measured by roll, pitch, and yaw angles relative to Earth-Centered Inertial (ECI) frame after solving problem summarized in Table 5.3.

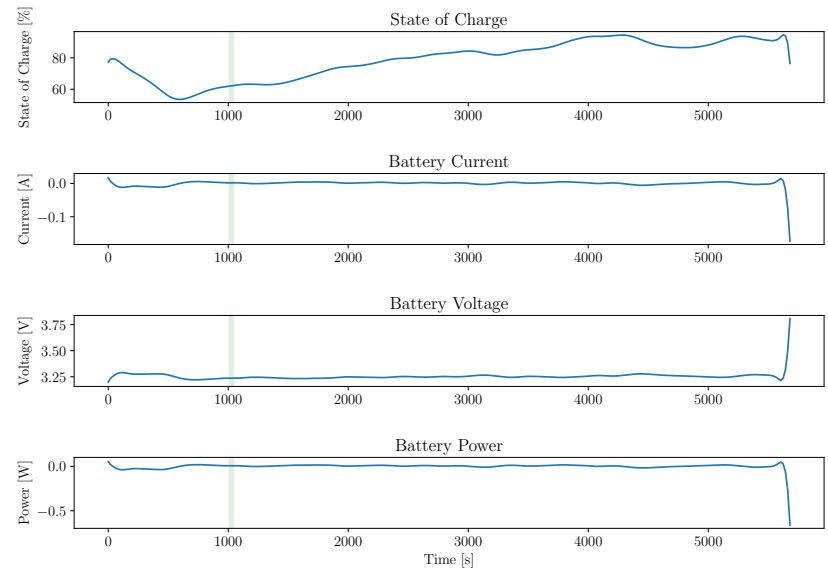


(a)

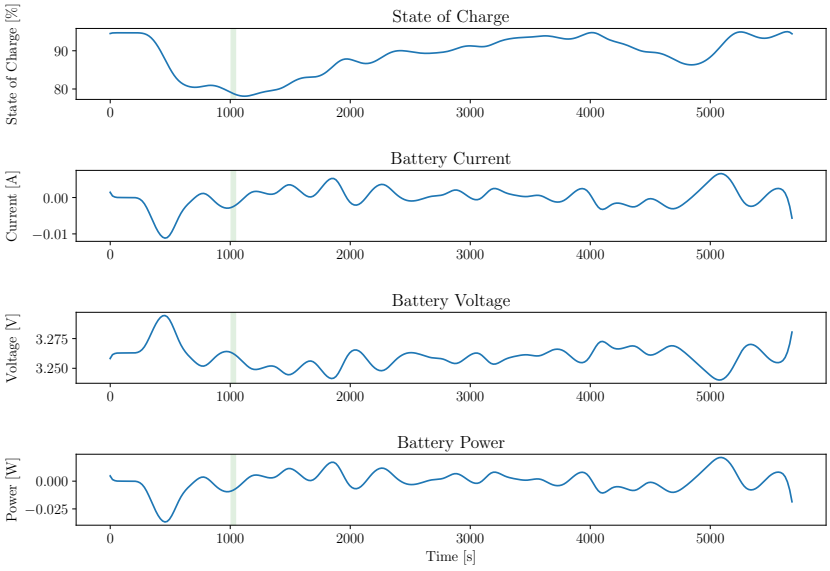


(b)

Figure 5.19. Percent of light available for each telescope component for: (a) optics and (b) detector spacecraft after solving problem summarized in Table 5.3. As the spacecraft points away from the Sun or enters Earth's shadow, the exposure to the Sun decreases.



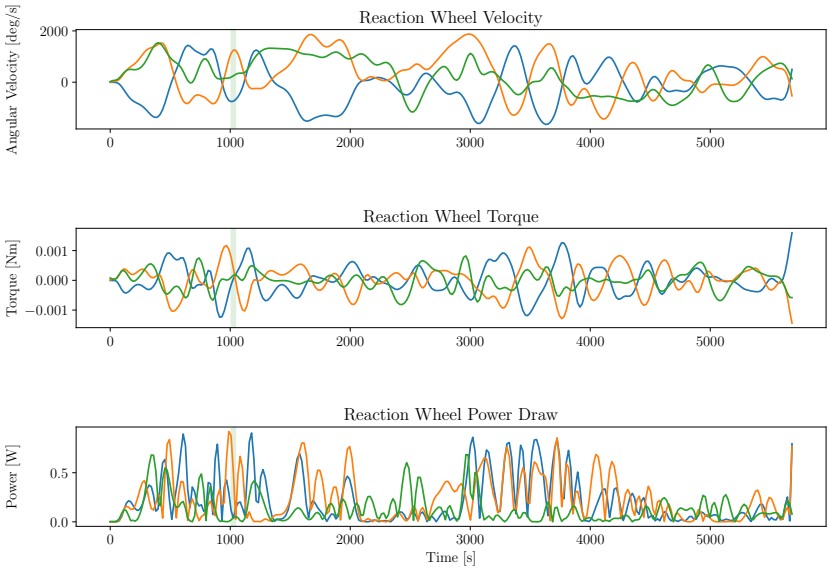
(a)



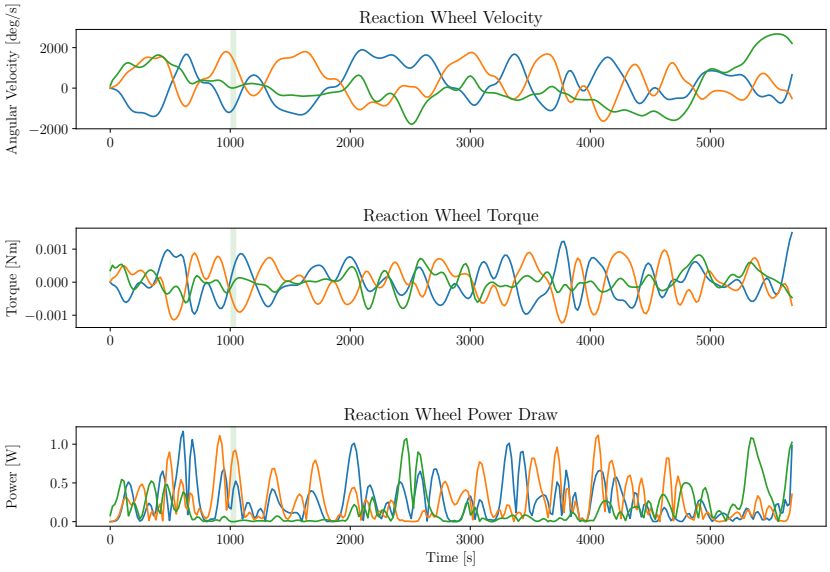
(b)

Figure 5.20. Battery state of charge, current, voltage, and power for: (a) optics and (b) detector spacecraft after solving problem summarized in Table 5.3.

Figure 5.22 shows the state of the solar arrays for each spacecraft. The solar array voltage is nearly identical for both spacecraft except during the observation phase where there is a



(a)



(b)

Figure 5.21. Reaction wheel speed, torque, and power for: (a) optics and (b) detector spacecraft after solving problem summarized in Table 5.3.

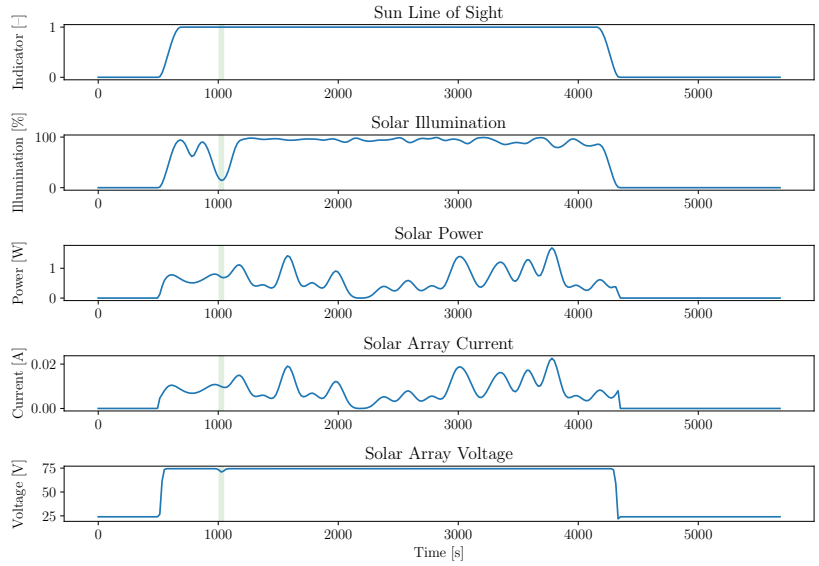
slight dip for the OSC. The solar illumination is also similar for both spacecraft except during the observation phase. The solar array voltage and solar illumination of the OSC solar arrays

drop during the observation phase due to the fact that the optics and the detector are oriented differently relative to the solar arrays on each spacecraft, requiring a different orientation during the observation phase for each spacecraft. The solar illumination of the DSC solar arrays follows a similar trend as the exposure of the detector instrument in Figure 5.19 (b) because the detector instrument is oriented in the same direction as the direction normal to the solar arrays.

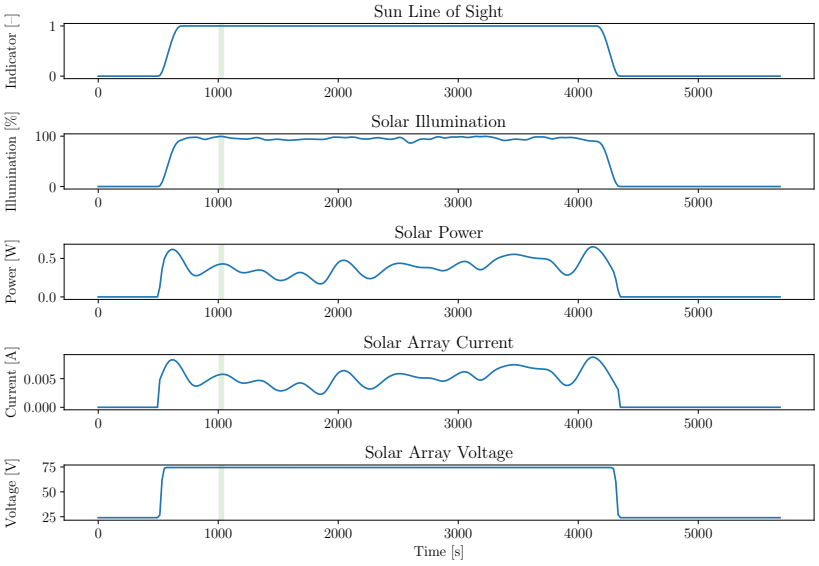
5.7 Conclusion

The objective of this study was to apply multidisciplinary design optimization (MDO) to the design of a virtual telescope comprised of two CubeSats. I used gradient-based optimization to optimize the trajectory, attitude profile, and battery sizing for each spacecraft. The solution relied on constraint aggregation and B-spline interpolation to reduce the size of the optimization problems. The key challenge was to satisfy constraints to millimeter-level accuracy when the positions of the CubeSats are on the order of thousands of kilometers. These large differences in orders of magnitude introduced truncation errors that render numerical optimization impossible. To enable numerical optimization, this chapter contributes a relative orbit model to reduce the relative magnitudes of intermediate state variables in the model. I find that the relative orbit model enables numerical gradient-based optimization while also taking into account orbit perturbations over multiple time scales.

The MDO problem was solved using a sequential approach, first solving the trajectory optimization problem for both spacecraft, and then using the solution in a subsequent optimization problem that simultaneously optimizes the attitude profile and battery pack. The successful design of a virtual telescope mission demonstrates MDO as a practical, scalable approach to solving spacecraft mission design problems involving swarms flying in formation. I find that a sequential approach to solving the original problem has robust convergence, while the approach that optimizes all disciplines simultaneously does not. In practice, aggregating constraints using a KS function (5.1) modified such that $\mathrm{KS}(kg(x))/k$ is used with $\rho = 1$, where k is a large number



(a)



(b)

Figure 5.22. Line of sight indicator and solar array illumination, power, current, and voltage for: (a) optics and (b) detector spacecraft after solving problem summarized in Table 5.3

appears to improve robustness across all optimization problems presented here.

Future work can take multiple directions. First, applying the methods developed in this

chapter to a swarm with a greater number of spacecraft will demonstrate the effectiveness of the orbit model developed in this chapter. Second, increasing the number and fidelity of spacecraft subsystem disciplines will demonstrate the robustness of the approach used in this chapter to changes in spacecraft configuration. Third, combining the trajectory, attitude profile, and battery sizing problems into one optimization problem and solving it simultaneously may lead to a better local optimum, although convergence remains an obstacle.

5.8 Acknowledgments

This work was supported in part by the National Science Foundation under grant no. 1936557.

Chapter 5, in full, is currently being prepared for submission of the material. Gandarillas V, Hwang JT; American Institute of Aeronautics and Astronautics, 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion

6.1 Summary of Contributions

The objectives of this dissertation were to: (1) apply a three-stage compiler design to the process of automatically performing efficient and accurate computation of derivatives for the class of models present in MDO problems; (2) measure the effect of using a declarative paradigm to describe models used in MDO problems on user code complexity; (3) measure the effect of the model structure on the run-time memory complexity of the software used to evaluate the model; and (4) apply MDO to the problem of designing a space-based virtual telescope using a model description amenable to gradient-based optimization. In this section, I summarize the contributions made by my research and their implications for the broader field of MDO.

My first contribution is a graph-based methdology for automatically generating software for evaluating models and automating adjoint-based sensitivity analysis. The graph-based methdology addresses the need to automate efficient and accurate derivative computation for the class of problems encountered in MDO through the application of a three-stage compiler to code-generation for models used in MDO problems. The three-stage compiler constructs a graph representation of the model based on whatever language is compatible with the compiler, and free to generate software to evaluate the model and compute sensitivities in any low-level language. The three-stage compiler uses the graph representation to enable efficient and accurate sensitivity analysis, and perform implementation-independent optimizations to improve run-time

performance.

My second contribution is the Computational System Design Language (CSDL), a declarative language that enables users to provide model descriptions rather than software implementations to a computer when solving MDO problems. CSDL addresses the need to remove obstacles to software implementation. Comparing the number of lines of code written in CSDL to the number of lines of code in other languages and frameworks (e.g. OpenMDAO) measures the effect of using a declarative language on code complexity and quantifies the impact of automating sensitivity analysis. CSDL was developed to separate model description from software implementation and yielded a twofold reduction in model code complexity over the OpenMDAO architecture. The measurable reduction in user code complexity and separation of model description from software implementation in the MDO workflow allows researchers and practitioners to focus on modeling and analysis without being burdened by the intricacies of software implementation, potentially reducing overall development time for large-scale design problems.

My third contribution is an approach for measuring the effect of the structure of the graph representation on potential run-time memory complexity and performance. This approach addresses the need to gain insight into the physical system's behavior and the program's run-time behavior from the model structure. The three-stage compiler developed in support of the first contribution uses a graph representation of the model structure. Analysis of the graph representation used within the three-stage compiler enables measuring the potential impact of the model structure on run-time memory complexity and performance. Users of the three-stage compiler can perform this analysis on their models regardless of the language used to describe the model or the compiler back end used to generate software to evaluate the model and perform sensitivity analysis. These measurements also inform the implementation strategy for generating software to evaluate the model, independent of the language used to describe the model.

My fourth contribution is a spacecraft model formulation that enables gradient-based MDO of spacecraft swarms with millimeter-level formation constraints. This contribution fills

a gap within the need to reformulate model descriptions by applying MDO to the design of a space-based virtual telescope, demonstrating the practical application and efficacy of the proposed methodologies developed in the first three contributions. In this problem, the discipline modeling orbital dynamics could be modeled in such a way so as to introduce numerical difficulties, rendering gradient-based optimization impossible. I also found that solving a series of optimization problems sequentially has robust convergence, while the approach that optimizes all disciplines simultaneously does not.

6.2 Recommendations for Future Work

In this section, I outline recommendations for future work given the contributions outlined in Section 6.1. First, I recommend continuing to explore developing CSDL as a mainly functional programming language, as the functional programming paradigm has not been explored for MDO languages outside of CSDL. Declarative paradigms such as functional programming make simpler expressions, independent of underlying implementations, the default. For example, CSDL currently enforces the rule that relationships defining each variable must be expressed once and only once; i.e., variables are immutable. This design forces the user to describe a model that resembles the original mathematical expressions that would appear in a paper, effectively bypassing a "translation layer" between mathematical notation and procedural syntax. Ironically, removing this "translation layer" has created a learning curve for new users of CSDL. Fortunately this immutability lifts from users the burden of defining unnecessary implementations and forces users to operate in the physical, rather than the computational domain. I recommend that any changes to the design of CSDL adhere to the philosophy of using a declarative paradigm to simplify model descriptions as much as possible. Furthermore, I encourage researchers in programming languages and human-computer interaction to investigate the impact of CSDL on MDO workflows. This includes studying the effect of language design choices on the speed of model development for individuals and teams.

Second, I recommend developing algorithms that leverage the structure of a graph representation to improve run-time memory complexity and performance. For example, taking advantage of dependency information within the graph to parallelize computation would lead to greater performance. Identifying the type of optimization problem will enable recommending or automatically choosing for the user the solver that will perform best on a given optimization problem.

Third, I recommend exploring the use of graph representations of models to gain insight into both the physical behavior of a system and the behavior of the software that evaluates the model, and using these insights to further automate processes within the MDO workflow. More implementation-independent optimizations could be developed to improve efficiency. For example, graph search algorithms could identify the type of optimization problem and inform users to choose an optimizer that performs well on that type of problem.

Bibliography

[1] V. Gandarillas and J. T. Hwang, "Talos: A toolbox for spacecraft conceptual design," *arXiv preprint arXiv:2303.14936*, 2023.

[2] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola, *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

[3] R. Rosen, *Life itself: a comprehensive inquiry into the nature, origin, and fabrication of life*. Columbia University Press, 1991.

[4] J. T. Hwang and J. R. Martins, "A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 4, pp. 1–39, 2018.

[5] J. Gray, K. Moore, and B. Naylor, "Openmdao: An open source framework for multidisciplinary analysis and optimization," in *13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*, p. 9101, 2010.

[6] W. E. Hart, J.-P. Watson, and D. L. Woodruff, "Pyomo: modeling and solving mathematical programs in python," *Mathematical Programming Computation*, vol. 3, no. 3, pp. 219–260, 2011.

[7] J. S. Gray, J. T. Hwang, J. R. Martins, K. T. Moore, and B. A. Naylor, "Openmdao: An open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, vol. 59, no. 4, pp. 1075–1104, 2019.

[8] M. L. Bynum, G. A. Hackebeil, W. E. Hart, C. D. Laird, B. L. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff, *Pyomo–optimization modeling in python*, vol. 67. Springer Science & Business Media, third ed., 2021.

[9] D. Sarojini, M. Ruh, A. Joshy, J. Yan, A. Ivanov, L. Scotzniovsky, A. Fletcher, N. Orndorff, M. Sperry, V. Gandarillas, I. Asher, J. Chambers, H. Gill, S. Lee, Z. Cheng, G. Rodriguez, S. Zhao, C. Mi, T. Nascenzi, T. Cuatt, T. Winter, A. Guibert, A. Cronk, H. Kim, S. Meng, and J. Hwang, "Large-scale multidisciplinary design optimization of an evtol aircraft using comprehensive analysis," in *AIAA SciTech 2023 Forum*, 2023.

[10] B. Wang, M. Sperry, V. E. Gandarillas, and J. T. Hwang, "Efficient uncertainty propagation through computational graph modification and automatic code generation," in *AIAA AVIATION 2022 Forum*, p. 3997, 2022.

[11] M. L. Ruh, D. Sarojini, A. Fletcher, I. Asher, and J. T. Hwang, "Large-scale multidisciplinary design optimization of the nasa lift-plus-cruise concept using a novel aircraft design framework," *arXiv preprint arXiv:2304.14889*, 2023.

[12] R. Swaminathan, D. Sarojini, and J. T. Hwang, "Integrating mbse and mdo through an extended requirements-functional-logical-physical (rflp) framework," in *AIAA AVIATION 2023 Forum*, p. 3908, 2023.

[13] B. Wang, N. C. Orndorff, M. Sperry, and J. T. Hwang, "High-dimensional uncertainty quantification using graph-accelerated non-intrusive polynomial chaos and active subspace methods," in *AIAA AVIATION 2023 Forum*, p. 4264, 2023.

[14] B. Wang, N. C. Orndorff, and J. T. Hwang, "Optimally tensor-structured quadrature rule for uncertainty quantification," in *AIAA SCITECH 2023 Forum*, p. 0741, 2023.

[15] M. Sperry, K. Kondap, and J. T. Hwang, "Automatic adjoint sensitivity analysis of models for large-scale multidisciplinary design optimization," in *AIAA AVIATION 2023 Forum*, p. 3721, 2023.

[16] V. E. Gandarillas and J. T. Hwang, "Large-scale multidisciplinary design optimization of a virtual-telescope cubesat swarm," in *AIAA AVIATION 2023 Forum*, p. 4021, 2023.

[17] E. J. Cramer, J. E. Dennis, Jr, P. D. Frank, R. M. Lewis, and G. R. Shubin, "Problem formulation for multidisciplinary optimization," *SIAM Journal on Optimization*, vol. 4, no. 4, pp. 754–776, 1994.

[18] N. M. Alexandrov, M. Y. Hussaini, *et al.*, *Multidisciplinary design optimization: State of the art*. SIAM, 1997.

[19] J. T. Hwang, J. P. Jasa, and J. R. Martins, "High-fidelity design-allocation optimization of a commercial aircraft maximizing airline profit," *Journal of Aircraft*, vol. 56, no. 3, pp. 1164–1178, 2019.

[20] J. T. Hwang and A. Ning, "Large-scale multidisciplinary optimization of an electric aircraft for on-demand mobility," in *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, p. 1384, 2018.

[21] T. H. Ha, K. Lee, and J. T. Hwang, "Large-scale multidisciplinary optimization under uncertainty for electric vertical takeoff and landing aircraft," in *AIAA Scitech 2020 Forum*, 2020.

[22] J. Hwang, D. Y. Lee, J. Cutler, and J. Martins, "Large-scale mdo of a small satellite using a novel framework for the solution of coupled systems and their derivatives," in *54th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, p. 1599, 2013.

[23] J. T. Hwang, D. Y. Lee, J. W. Cutler, and J. R. Martins, "Large-scale multidisciplinary optimization of a small satellite's design and operation," *Journal of Spacecraft and Rockets*, vol. 51, no. 5, pp. 1648–1663, 2014.

[24] W. Squire and G. Trapp, "Using complex variables to estimate derivatives of real functions," *SIAM review*, vol. 40, no. 1, pp. 110–112, 1998.

[25] J. R. Martins, P. Sturdza, and J. J. Alonso, "The complex-step derivative approximation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, no. 3, pp. 245–262, 2003.

[26] J. R. Martins and J. T. Hwang, "Review and unification of methods for computing derivatives of multidisciplinary computational models," *AIAA journal*, vol. 51, no. 11, pp. 2582–2599, 2013.

[27] C. Marriage and J. R. Martins, "Reconfigurable semi-analytic sensitivity methods and mdo architectures within the pimdo framework," in *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, p. 5956, 2008.

[28] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 9, no. 4, p. e1305, 2019.

[29] C. C. Margossian and M. Betancourt, "Efficient automatic differentiation of implicit functions," *arXiv preprint arXiv:2112.14217*, 2021.

[30] B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt, "The stan math library: Reverse-mode automatic differentiation in c++," *arXiv preprint arXiv:1509.07164*, 2015.

[31] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *Journal of statistical software*, vol. 76, 2017.

[32] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

[33] J. Yang, "Fastad: Expression template-based c++ library for fast and memory-efficient automatic differentiation," 2021.

[34] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert, "Efficient and modular implicit differentiation," 2022.

[35] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Marchine Learning Research*, vol. 18, pp. 1–43, 2018.

[36] J. Kallrath, *Modeling Languages in Mathematical Optimization*. Springer Science & Business Media, 2004.

[37] J. Kallrath, *Algebraic Modeling Systems: Modeling and Solving Real World Optimization Problems*, vol. 104. Springer Science & Business Media, 2012.

[38] I. Dunning, J. Huchette, and M. Lubin, "Jump: A modeling language for mathematical optimization," *SIAM review*, vol. 59, no. 2, pp. 295–320, 2017.

[39] R. Fourer, D. M. Gay, and B. W. Kernighan, "A modeling language for mathematical programming," *Management Science*, vol. 36, no. 5, pp. 519–554, 1990.

[40] M. R. Bussieck and A. Meeraus, "General algebraic modeling system (gams)," in *Modeling languages in mathematical optimization*, pp. 137–157, Springer, 2004.

[41] J. Bisschop and A. Meeraus, "On the development of a general algebraic modeling system in a strategic planning environment," in *Applications*, pp. 1–29, Springer, 1982.

[42] J. Åkesson, "Optimica—an extension of modelica supporting dynamic optimization," in *6th International Modelica Conference, 2008*, 2008.

[43] P. A. Fishwick, "A taxonomy for simulation modeling based on programming language principles," *IIE transactions*, vol. 30, no. 9, pp. 811–820, 1998.

[44] R. Sinha, C. J. Paredis, V.-C. Liang, and P. K. Khosla, "Modeling and simulation methods for design of engineering systems," *J. Comput. Inf. Sci. Eng.*, vol. 1, no. 1, pp. 84–91, 2001.

[45] L. Viklund and P. Fritzson, "Objectmath–an object-oriented language and environment for symbolic and numerical processing in scientific computing," *Scientific Programming*, vol. 4, no. 4, pp. 229–250, 1995.

[46] P. Fritzson and V. Engelson, "Modelica—a unified object-oriented language for system modeling and simulation," in *European Conference on Object-Oriented Programming*, pp. 67–90, Springer, 1998.

[47] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 2, pp. 1–37, 2014.

[48] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[49] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," *arXiv preprint arXiv:1607.07892*, 2016.

[50] R. C. Kirby and A. Logg, "A compiler for variational forms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 3, pp. 417–444, 2006.

[51] M. E. Rognes, R. C. Kirby, and A. Logg, "Efficient assembly of h(div) and h(curl) conforming finite elements," *SIAM Journal on Scientific Computing*, vol. 31, no. 6, pp. 4130–4151, 2010.

[52] K. B. Ølgaard and G. N. Wells, "Optimizations for quadrature representations of finite element tensors through automated code generation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 1, pp. 1–23, 2010.

[53] A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Ffc: the fenics form compiler," in *Automated Solution of Differential Equations by the Finite Element Method*, pp. 227–238, Springer, 2012.

[54] M. S. Alnæs and K.-A. Mardal, "On the efficiency of symbolic computations combined with code generation for finite element methods," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 1, pp. 1–26, 2010.

[55] M. S. Alnæs and K.-A. Mardal, "Syfi and sfc: Symbolic finite elements and form compilation," in *Automated Solution of Differential Equations by the Finite Element Method*, pp. 273–282, Springer, 2012.

[56] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[57] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[58] W. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds.), vol. 33, pp. 12472–12485, Curran Associates, Inc., 2020.

[59] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert, "Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[60] W. S. Moses, S. H. K. Narayanan, L. Paehler, V. Churavy, M. Schanen, J. Hückelheim, J. Doerfert, and P. Hovland, "Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22, IEEE Press, 2022.

[61] A. Ning and T. McDonnell, "Automating steady and unsteady adjoints: Efficiently utilizing implicit and algorithmic differentiation," 2023.

[62] B. Wang, M. Sperry, V. E. Gandarillas, and J. T. Hwang, "Accelerating model evaluations in uncertainty propagation on tensor grids using computational graph transformations," *AIAA Journal*, (under review).

[63] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[64] J. P. Jasa, J. T. Hwang, and J. R. R. A. Martins, "Open-source coupled aerostructural optimization using Python," *Structural and Multidisciplinary Optimization*, vol. 57, pp. 1815–1827, April 2018.

[65] S. S. Chauhan and J. R. R. A. Martins, "Low-fidelity aerostructural optimization of aircraft wings with a simplified wingbox model using OpenAeroStruct," in *Proceedings of the 6th International Conference on Engineering Optimization, EngOpt 2018*, (Lisbon, Portugal), pp. 418–431, Springer, 2018.

[66] M. L. Ruh and J. T. Hwang, "Robust modeling and optimal design of rotors using blade element momentum theory," in *AIAA AVIATION 2021 FORUM*, p. 2598, 2021.

[67] Z. Cheng, S. Zhao, L. Scotzniovsky, G. Rodriguez, C. Mi, and J. Hwang, "A differentiable method for low-fidelity analysis of permanent-magnet synchronous motors," in *AIAA SciTech 2023 Forum*, (in review).

[68] W. H. Swartz, S. R. Lorentz, P. M. Huang, S. M. Reilly, N. M. Reilly, and S. J. Papadakis, "Radiometer assessment using vertically aligned nanotubes (ravan)," in *IGARSS 2019-2019 IEEE International Geoscience and Remote Sensing Symposium*, pp. 4975–4977, IEEE, 2019.

[69] B. A. Cohen, P. O. Hayne, B. Greenhagen, D. A. Paige, C. Seybold, and J. Baker, "Lunar flashlight: Illuminating the lunar south pole," *IEEE Aerospace and Electronic Systems Magazine*, vol. 35, no. 3, pp. 46–52, 2020.

[70] D. McCammon and P. Kaaret, "Xqcsat: A high-resolution spectroscopic study of hot gas in the halo and interstellar medium," in *American Astronomical Society Meeting Abstracts*, vol. 234, pp. 108–05, 2019.

[71] P. Kaaret, D. McCammon, T. Nguyen, and D. Frank, "Xqcsat-x-ray quantum calorimeter satellite," *The Space Astrophysics Landscape for the 2020s and Beyond*, vol. 2135, p. 5007, 2019.

[72] G. Krieger, A. Moreira, H. Fiedler, I. Hajnsek, M. Werner, M. Younis, and M. Zink, "Tandem-x: A satellite formation for high-resolution sar interferometry," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 45, no. 11, pp. 3317–3341, 2007.

[73] R. Burns, C. A. McLaughlin, J. Leitner, and M. Martin, "Techsat 21: formation design, control, and simulation," in *2000 IEEE Aerospace Conference. Proceedings (Cat. No. 00TH8484)*, vol. 7, pp. 19–25, IEEE, 2000.

[74] A. Cheng, P. Michel, C. Reed, A. Galvez, I. Carnelli, and P. Headquarters, "Dart: Double asteroid redirection test," in *European Planetary Science Congress*, vol. 7, pp. 23–28, 2012.

[75] A. S. Rivkin, N. L. Chabot, A. M. Stickle, C. A. Thomas, D. C. Richardson, O. Barnouin, E. G. Fahnestock, C. M. Ernst, A. F. Cheng, S. Chesley, *et al.*, "The double asteroid redirection test (dart): Planetary defense investigations and requirements," *The Planetary Science Journal*, vol. 2, no. 5, p. 173, 2021.

[76] B. Macintosh, S. D'Amico, A. Koenig, and A. Madurowicz, "Miniature distributed occulter telescope (mdot): A concept for a smallsat mission to observe extrasolar zodiacal dust and exoplanets," in *AGU Fall Meeting Abstracts*, vol. 2019, pp. A43F–08, 2019.

[77] R. Pirayesh, A. Naseri, F. Moreu, S. Stochaj, N. Shah, and J. Krizmanic, "Attitude control optimization of a two-cubesat virtual telescope in a highly elliptical orbit," in *Space Operations: Inspiring Humankind's Future*, pp. 233–258, Springer, 2019.

[78] K. Rankin, S. Stochaj, N. Shah, J. Krizmanic, and A. Naseri, "Vtxo: Virtual telescope for x-ray observations," *9th International Workshop on Satellite Constellations and Formation Flying*, p. 11, June 2017.

[79] H. Park, Y. Lee, Y. Lee, K. Rankin, and S. Stochaj, "Design optimization of a space virtual telescope mission using a cubesat swarm," in *70th International Astronautical Congress, Washington D.C.*, 2018.

[80] J. S. Llorente, A. Agenjo, C. Carrascosa, C. de Negueruela, A. Mestreau-Garreau, A. Cropp, and A. Santovincenzo, "Proba-3: Precise formation flying demonstration mission," *Acta Astronautica*, vol. 82, no. 1, pp. 38–46, 2013.

[81] D. Galano, D. Jollet, K. Mellab, J. Villa, L. Penin, R. Rougeot, and J. Versluys, "Proba-3 precise formation flying mission," in *International Workshop on Satellites Contellations and Formation Flying (IWSCFF), Glasgow, United Kingdom*, 2019.

[82] E. G. Lightsey, E. Arunkumar, E. Kimmel, M. Kolhof, A. Paletta, W. Rawson, S. Selvamurugan, J. Sample, T. Guffanti, T. Bell, *et al.*, "Concept of operations for the visors mission: A two satellite cubesat formation flying telescope," in *AAS 22-125, 2022 AAS Guidance, Navigation and Control Conference*, 2022.

[83] A. Gundamraj, R. Thatavarthi, C. Carter, E. G. Lightsey, A. Koenig, and S. D'Amico, "Preliminary design of a distributed telescope cubesat formation for coronal observations," in *AIAA Scitech 2021 Forum*, p. 0422, 2021.

[84] A. Koenig, S. D'Amico, and E. G. Lightsey, "Formation flying orbit and control concept for the visors mission," in *AIAA Scitech 2021 Forum*, p. 0423, 2021.

[85] T. Mosher, "Applicability of selected multidisciplinary design optimization methods to conceptual spacecraft design," in *6th Symposium on Multidisciplinary Analysis and Optimization*, p. 4052, 1996.

[86] J. Guo, Z. You, and D. Ren, "A distributed multidisciplinary design optimazation architecture for spacecraft design," in *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, p. 5481, 2002.

[87] W. Wu, H. Huang, S. Chen, and B. Wu, "Satellite multidisciplinary design optimization with a high-fidelity model," *Journal of Spacecraft and Rockets*, vol. 50, no. 2, pp. 463–466, 2013.

[88] J. R. Martins and A. B. Lambe, "Multidisciplinary design optimization: a survey of architectures," *AIAA journal*, vol. 51, no. 9, pp. 2049–2075, 2013.

[89] E. R. Taylor, "Evaluation of multidisciplinary design optimization techniques as applied to spacecraft design," in *2000 IEEE Aerospace Conference. Proceedings (Cat. No. 00TH8484)*, vol. 1, pp. 371–384, IEEE, 2000.

[90] V. Gandarillas, A. J. Joshy, M. Z. Sperry, A. K. Ivanov, and J. T. Hwang, "A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis," *Structural and Multidisciplinary Optimization*, (under review).

[91] N. M. Poon and J. R. Martins, "An adaptive approach to constraint aggregation using adjoint sensitivity analysis," *Structural and Multidisciplinary Optimization*, vol. 34, no. 1, pp. 61–73, 2007.

[92] P. E. Gill, W. Murray, and M. A. Saunders, "SNOPT: An SQP algorithm for large-scale constrained optimization," *SIAM Rev.*, vol. 47, pp. 99–131, 2005.

[93] J. T. Hwang and J. R. Martins, "A fast-prediction surrogate model for large datasets," *Aerospace Science and Technology*, vol. 75, pp. 74–87, 2018.

[94] Blue Canyon Technologies, *Reaction Wheels*, 2022.

[95] J. Hwang, "Efficient and scalable computational design of a small satellite," 2014.

[96] H. Kawamura, K. Naka, N. Yonekura, S. Yamanaka, H. Kawamura, H. Ohno, and K. Naito, "Simulation of i–v characteristics of a pv module with shaded pv cells," *Solar Energy Materials and Solar Cells*, vol. 75, no. 3-4, pp. 613–621, 2003.

[97] M.-K. Tran, M. Mathew, S. Janhunen, S. Panchal, K. Raahemifar, R. Fraser, and M. Fowler, "A comprehensive equivalent circuit model for lithium-ion batteries, incorporating the effects of state of health, state of charge, and temperature on model parameters," *Journal of Energy Storage*, vol. 43, p. 103252, 2021.

[98] A. B. Lambe and J. R. Martins, "Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes," *Structural and Multidisciplinary Optimization*, vol. 46, pp. 273–284, 2012.