# Massachusetts Institute of Technology
# Department of Aeronautics and Astronautics


## Thesis Proposal
## Doctor of Philosophy

# Verification-guided design for autonomous systems

Date of Submission:

June 6, 2023


Author: Charles Dawson
PhD Candidate

Committee: Chuchu Fan (Chair)
Russ Tedrake
Sertaç Karaman

External Evaluator: Vikash Mansinghka

# Contents

## Abstract

Abstract should be no more than **300 words in 1 page**.

State the significance of the proposed research. Include long-term objectives and specific aims. Describe concisely the research design and methods for achieving these objectives. Highlight the specific hypotheses to be tested, goals to be reached, or technology to be developed, which are intended to be your original contributions. Avoid summaries of past accomplishments.

# 1 Introduction & Objectives

Before robots can be deployed in safety-critical environments, we must be able to verify they they will perform safely. In cases where real-world testing is too risky or expensive, robotics engineers must instead rely on mathematical modeling or simulation to verify that a robot will perform as intended. Unfortunately, there are issues with each of these approaches. Although mathematical models are amenable to formal proofs, robots are often too complex to reduce to a set of equations. On the other hand, although simulators can handle the full complexity of a robotic system, they are often treated as black-boxes, providing an incomplete view of a robot's performance. In order to safely deploy complex robots in the real world, we require new tools that blend the rigor of mathematical modeling with the scalability and generality of simulation.

In my thesis, I aim to close the gap between formal methods based on mathematical models [Belta and Sadraddini, 2019, Kress-Gazit et al., 2018] and simulation-based verification techniques [Zhou et al., 2021, Corso et al., 2021, O' Kelly et al., 2018] to develop tools to help engineers more easily *design* and *verify* complex robotic systems. My key insight is that simulators, as computer programs, are not black boxes but instead contain rich mathematical structure. If we can exploit this structure using program analysis tools (e.g. automatic differentiation, tracing, etc.), we can combine the flexibility of simulation with the rigor of mathematical reasoning. Using this insight, I aim to develop tools to support the *design-analysis cycle* for robots and other safety-critical cyberphysical systems in three ways:

1. *Design optimization:* automatically search for design parameters that achieve good performance.

2. *Safety verification:* characterize the robustness of a design and predict corner cases where it is likely to fail (either by violating a constraint or incurring a high cost).

3. *Verification-guided design:* closing the feedback loop between verification and design, e.g. by using predicted corner cases to guide future design iterations.

These tools will support engineers in developing increasingly complex robotic systems, enabling a more efficient design process and providing the ability to verify the safety of a design *before* deployment. Moreover, I will apply my research not just to traditional robotic systems, but to cyberphysical systems such as large-scale energy, transportation, and industrial networks.

## 1.1 Impact

Recent years have seen large numbers of learning-enabled autonomous systems deployed in the real world. Unfortunately, increased deployment has seen a corresponding increase in accidents involving these systems. More and more, the development and deployment of autonomous systems risks being stalled due to public and regulatory concern over safety [Posada, ]. Unlike other forms of safety-critical infrastructure like bridges and buildings, autonomous systems suffer from both *complexity*, leading to difficult-to-predict emergent behavior resulting from interactions between subsystems, and *uncertainty* stemming from the environment or the actions of other agents. While engineers in other fields have access to a range of computer-aided tools to boost their productivity and ensure the safety of their designs, such as computer-aided design (CAD) and finite-element analysis (FEA) in mechanical and civil engineering or electronic design automation (EDA) in electrical engineering, robotics engineers lack such comprehensive design and verification tools. In the

abscence of these tools, engineers must rely on an ad-hoc process relying heavily on experience and tedious parameter tuning. In my thesis, I aim to address this gap by developing efficient computational tools to assist engineers in designing and verifying safety-critical autonomous systems, with the goal of helping engineers more confidently *design*, *debug*, and *deploy* autonomous systems.

## 1.2 Outline

The rest of this document is structured as follows. Section 2 provides the necessary background and reviews related work with an eye towards highlighting the significance of my proposed thesis contributions. Section 3 presents progress that I have already made towards my thesis objectives by designing methods that solve verification-guided design problems using local optimization with gradients derived from differentiable simulation. These methods demonstrate good performance on a range of robotics and control problems, but they rely on local methods that do not give a complete picture of system safety. Section 4 presents in-progress work that addresses this issue by using approximate Bayesian inference rather than optimization, which allows for more exploration of the design/failure search spaces and are more robust to inaccuracies in automatically-computed gradients. Section 5 describes the remainder of the research I plan to include in my thesis, and Section 6 includes my proposed thesis timeline and other logistical details.

# 2 Background and Significance

My thesis aims to build on prior work on safety verification (both model-based and black-box) and program analysis (in the form of differentiable and probabilistic programming). This section will review each of these fields with an eye towards framing the significance of my planned thesis contributions.

## 2.1 Safety verification

Safety and robustness are critical concerns for any robotic system, and there is a correspondingly large body of work studying how to verify safety properties for autonomous systems. These approaches can be broadly categorized into *model-based* and *model-free* works. Model-based approaches rely on a mathematical model of the system to be verified, and use this model to prove that the system satisfies the desired safety properties. Model-free approaches, on the other hand, do not require a mathematical model of the system, and instead use a large number of samples of the system's input-output behavior to characterize its safety.

Regardless of approach, verification methods typically aim to solve one of three problems [Corso et al., 2021]: falsification, likely failure analysis, or failure probability estimation. Denote the disturbance space $y \in \mathcal{Y} \subseteq \mathbb{R}^n$, a simulator $f \colon \mathcal{Y} \to \Xi$ that maps disturbances to system traces $\xi$, and a specification $\psi \colon \Xi \to \mathbb{R}$ that determines whether a trace satisfies ($\psi(\xi) \geq 0$) or does not satisfy ($\psi(\xi) < 0$) the safety constraints. In this context, the falsification problem involves solving an optimization for a counterexample

$$\text{find } y \in \mathcal{Y} \text{ s.t. } \psi(f(y)) < 0 \tag{2.1}$$

which is commonly reparameterized as a minimization problem to find the worst counterexample

$$\min_{y \in \mathcal{Y}} \psi(f(y)) \tag{2.2}$$

The likely failure analysis problem adds information about the prior distribution of disturbances $p(y)$ and aims to find the most likely counterexample

$$\max_{y \in \mathcal{Y}} p(y) \text{ s.t. } \psi(f(y)) < 0 \tag{2.3}$$

Finally, the failure probability estimation problem aims to estimate the probability that the system violates the safety constraints

$$\Pr\left[\psi(f(y)) < 0\right] = \mathbb{E}_{y \sim p(y)}\left[\mathbb{1}\{\psi(f(y))\}\right] \tag{2.4}$$

### 2.1.1 Model-based safety verification

Early approaches to model-based verification and fault identification used symbolic logical models of the system under test to formally reason about failures using (computationally expensive) satisfiability (SAT) solvers or search [de Kleer and Williams, 1987, Benard et al., 2000]. More recent approaches to model-based failure mode identification have used mathematical models of the system dynamics to frame the problem through the lens of reachability [Annpureddy et al., 2011a,Bansal et al., 2017], optimal control [Chou et al., 2018], or optimization (e.g. sum-of-squares, or SoS [Ahmadi and Majumdar, 2016, Majumdar et al., 2014]).

The primary challenge facing all of these methods is that it may be difficult or impossible to construct a symbolic model for the system under test. For example, simulating the dynamics of a power transmission system or certain contact models requires solving an optimization problem and does not have a closed form. Even when it is theoretically possible to obtain a closed-form symbolic model, in practice the need to construct and manipulate large sets of equations introduces the possibility of error and requires a large amount of human effort. Historically, this difficulty motivated the development of model-free approaches to safety verification.

### 2.1.2 Black-box safety verification

In practice, although we may not have access to a mathematical or symbolic model of a system, we often have access to a simulator instead, motivating a set of so-called "black-box" methods. These methods are characterized by restricting the verification algorithm to sampling input-output pairs from the simulator without side information such as gradients [Corso et al., 2021]. Since black-box methods are usually quite easy to integrate with an existing simulator (often relying on a standardized API such as the OpenAI Gym interface [Brockman et al., 2016]), they have seen widespread use, particularly in verification for autonomous vehicles [Xu et al., 2022a, Riedmaier et al., 2020, O' Kelly et al., 2018, Corso et al., 2019, Wang et al., 2021, Sun et al., 2021, Zhong et al., 2022, Corso and Kochenderfer, 2020, Zhang et al., 2022, Hanselmann et al., 2022]. The three most common types of black-box verification method involve black-box optimization, reinforcement learning, or black-box inference methods [Corso et al., 2021].

**Black-box optimization** uses the optimization formulations in Eq. (2.2) or Eq. (2.3) to search for a set of inputs that violate the safety property of interest, using methods like Bayesian optimization [Wang et al., 2021], REINFORCE [Ding et al., 2020], and ant colony optimization [Annpureddy et al., 2011b]. There are a large number of subtly-different black-box optimization schemes that could be applied to this problem [Kochenderfer and Wheeler, 2019], all making different tradeoffs between exploration and exploitation.

**Reinforcement learning** break the monolithic optimization problem in Eq. (2.2) into a sequential decision making problem by simulating single steps rather than entire trajectories, then optimizing a policy that maximizes a reward that encourages violation of the safety property [Corso et al., 2019]. This reward is often hand-designed using domain expertise, e.g. by rewarding an adversarial vehicle for reducing the distance between it and the vehicle under test [Ding et al., 2020].

**Black-box inference** takes inspiration from algorithms for approximate Bayesian inference and are most often applied to the failure probability estimation [O' Kelly et al., 2018] and likely failure mode problems [Zhou et al., 2021]. To estimate failure probability, [O' Kelly et al., 2018] uses adaptive importance sampling to guide exploration of the search space towards regions that are more likely to induce a failure (while adjusting the failure probability estimate to account for this biased exploration). To generate likely failure modes, [Zhou et al., 2021] uses gradient-free Markov chain Monte Carlo (MCMC) to sample failure modes that are likely to induce a failure.

### 2.1.3 Limitations and significance

In the context of prior work on model-based and black-box safety verification, this thesis aims to address two major technical gaps.

**Model-based approaches are difficult to scale to complex systems** The requirement of a symbolic model is either theoretically impossible (when closed-form models do not exist) or practically infeasible (for large-scale systems with multiple interacting subsystems). The difficulty of constructing symbolic models by hand motivates my work in this thesis, which seeks to automatically extract information about the mathematical structure of a system from the simulator using program analysis techniques (discussed in the next section).

**Black-box methods struggle with sample complexity** Lacking access to gradient information that could help guide their exploration of high-dimensional spaces, black-box methods often struggle to scale to high-dimensional problems, or (if they can scale) require a large number of simulated rollouts to converge. Although black-box methods are easy to integrate with existing simulators, these scalability and sample-complexity issues motivate my work in this thesis, which seeks to improve scalability and sample complexity by incorporating additional side information (e.g. automatically-derived gradients) into the optimization process.

## 2.2 Programs as mathematical models

Historically, symbolic models have been synonymous with hand derivation, a tedious and error-prone process that does not scale to complex systems. However, recent work in the programming languages community has shown that there is an alternate way to construct these models: by exploiting the rich mathematical structure available embedded in computer programs themselves.

In practice, often the term "black-box" is used to describe the setting where we have access to a computer program implementing a simulator of the system under test. The next two sections will show how we can obtain varying degrees of introspection into the structure of these programs using program analysis methods, effectively granting the ability to look inside the black box. In particular, we will discuss two exciting program analysis techniques that are relevant to this thesis: automatic differentiation (which treats computer programs as mathematical functions that can be differentiated) and probabilistic programming (which treats stochastic programs as graphical models that can be used for Bayesian inference).

### 2.2.1 Automatic differentiation

Perhaps the most well-known (and widely-used) program analysis method in the machine learning and robotics communities is automatic differentiation (autodiff, or AD). AD was developed in TODO-historical-note-and-citation, and achieved widespread use in the form of backpropagation for training neural networks todo-cite. The popularity of neural networks prompted the development of differentiable tensor math libraries such as PyTorch [Paszke et al., 2019], TensorFlow todo-cite, and JAX [Bradbury et al., 2018], and specialized AD tools followed in the form of differentiable optimization layers [Agrawal et al., 2019], simulators [Hu et al., 2019], renderers [Hu et al., 2019, Le Lidec et al., 2021], and even task specifications [Leung et al., 2021], to name but a few (the discussion in this document is largely Python-focused, but similar ecosystems exist in Julia, C++, and other LLVM languages todo-cite).

At a high level, the aim of automatic differentiation is to allow the user to implement some function $y = f(x) : \mathbb{R}^n \mapsto \mathbb{R}^m$ and then provide the ability, without writing any additional code, to obtain the Jacobian $Df(x) \in \mathbb{R}^{m \times n}$. This is typically done in one of two ways, referred to as forward- and reverse-mode AD, respectively. In the interest of space, this document will provide only a brief overview of these two methods; todo-cite-jax-page provides a more thorough introduction.

Forward-mode AD computes the product between a vector in the input tangent space $\delta x$ and the Jacobian, "pushing forward" into the tangent space of the output $\delta x \mapsto \delta y = Df(x)\delta x$. As a result, forward-mode AD is sometimes referred to as the Jacobian-vector product (JVP) or the pushforward map. It is typically implemented by operator overloading, in which primitive operations (e.g. $+$, $\times$, sin, etc.) are overloaded to operate on a new data type that carries both the primal $x$ and tangent $\delta x$. As the function is computed and each primitive operation is carried out, the tangent value is updated using hand-derived derivative rules for each operation. The benefit of forward-mode AD is that its memory usage is constant with respect to the number of operations used to define $f$ (roughly double the memory usage of just evaluating the function value). The potential downside is that computing the Jacobian requires one primal and tangent evaluation of $f$ for each column of the Jacobian; this is fine for functions with few inputs and many outputs, but does not scale to typical machine learning applications (where we often wish to differentiate with respect to thousands of model parameters).

Reverse-mode AD computes the product of a vector in the output tangent space $\delta y$ and the transposed Jacobian, "pulling back" into the tangent space of the input $\delta y \mapsto \delta x = Df(x)^T \delta y$, and it is referred to as the pullback map or vector-Jacobian product (VJP) accordingly. This mode is typically implemented as generalized backpropagation that tracks which primitive operations are applied in evaluating the output value $y = f(x)$ and constructs a computation graph. To compute the derivative, we trace backwards through the computation graph, applying derivative rules for each computation. The benefit of this method is that computing the full Jacobian requires one forward and backward pass for each row of the Jacobian, which is much more efficient in practice for systems with many input parameters and few outputs (including typical ML and optimization applications where there are many parameters or decision variables that yield a scalar objective output). The downside of this mode is that it is much more memory intensive, since the results of intermediate operations during the forward pass are typically cached and reused during the backwards pass, and the memory requirements scale linearly with the size of the function's computation graph.

Both of these modes require hand-derived derivative rules for the primitive operations used in computing $f$; however, in certain cases these rules may be overridden to provide more accurate or numerically stable gradients. An important case that commonly arises in robotics is the case where $f$ involves solving either an optimization (e.g. $f(x) = \arg\min_y g(y, x)$) or root-finding problem (e.g. $f(x) = \text{find}_y \text{ s.t. } g(y, x) = 0$). In practice, these problems are solved iteratively, but naïvely differentiating the primitive operations applied in each iteration is both costly and inaccurate. Instead, these optimization and root-finding problems are treated as primitive operations for the purposes of AD, and the derivative rules are found automatically using the implicit function theorem [Agrawal et al., 2019]. This approach allows for accurate differentiation of implicit dynamics (including certain contact models [Howell et al., 2022]), rendering (since raytracing is a root-finding procedure), and optimization-based control.

It is important to note that although the gradients derived from AD are often referred to as exact or analytic, they are still only estimates of the true gradient. Most commonly used AD systems are not sound, admitting pathological inputs that can yield arbitrarily wrong gradients

when differentiated (e.g. $f(x) = \{x \text{ if } x \neq 0;\ -x \text{ otherwise}\}$, which is identically equal to $y = x$ in both value and gradient but yields $df/dx(0) = -1$ when differentiated with most AD libraries). It is possible to detect some of these pathological cases, either at runtime or at compile-time, but even non-pathological functions can be stiff (with very large gradients), non-smooth, or even discontinuous. As a result, it is important to consider how these inaccuracies will affect downstream consumers of the gradient (e.g. for optimization). The downstream effects of these artifacts is an active area of research [Suh et al., 2021, Suh et al., 2022, Metz et al., 2022].

**Significance of proposed thesis** The last few years have seen a surge of applications of AD, including robotics problems such as control synthesis and system identification [Xu et al., 2022b, Agrawal et al., 2019, Amos and Kolter, 2017, de Avila Belbute-Peres et al., 2018, Du et al., 2021]. However, despite these successful applications, two important research questions remain to be answered in this thesis. First, since both modes of AD require more computation than a standard function evaluation, we must ask whether the derivative juice is worth the computational squeeze. That is, do AD-derived gradients provide enough of a performance increase on downstream tasks, relative to gradient-free optimizers or zero-order gradient estimates and evaluated on robotics-relevant benchmarks, to merit this additional computational expense? Second, given that AD can yield poorly-conditioned gradients on many problems of interest for robotics, can we design downstream algorithms that are robust to variance or inaccuracy in the gradients they receive?

### 2.2.2 Probabilistic programming

Differentiable programming, although popular, is just the tip of the program analysis iceberg. Another promising emerging field of program analysis is that of probabilistic programming, which extends its view to include programs that make random choices (e.g. by querying a random number generator). Probabilistic programming takes the perspective that the computation graph of such a program can be viewed as a graphical probabilistic model (e.g. Bayesian network) that encodes a joint distribution over variables involved in the computation todo cite survey or tutorial. With this mindset, we can gain an additional level of introspection into the behavior of a program by automatically deriving this graphical model and applying approximate Bayesian inference techniques; for instance conditioning on certain variables and estimating the posterior distribution.

It is important to note that probabilistic programming is not an alternative to AD (in fact, most probabilistic programming frameworks rely on AD). AD treats programs as mathematical functions that can be differentiated, and probabilistic programming assigns a semantic meaning to those derivatives in the context of probabilistic inference problems. Moreover, while it is relatively easy to port existing code to use AD (e.g. replacing `numpy` with `jax.numpy` in Python), probabilistic programming frameworks require additional annotation of the source code, often defining additional syntax to annotate random choices (so that such choices can be traced and referred to while solving inference problems).

Treating programs as probabilistic models offers a number of benefits. Both constrained and unconstrained optimization problems can be transcribed as posterior inference problems (this is the optimization-as-inference approach discussed in [Ma et al., 2019]todo-opt-as-inference-cite), but inference also allows us to answer questions such as the likely failure analysis problem in (2.3). In addition, the probabilistic mindset allows us to deal with models that combine continuous and discrete choices in a principled way (as hybrid continuous-discrete graphical models) and design inference algorithms accordingly todo-cite. Finally, probabilistic programming can also be applied

to program synthesis, since the structure of the program's computation graph can itself be seen as the output of a stochastic program-generating process, allowing us to use conditional inference to search over the space of programs that might solve a particular problem [Cusumano-Towner et al., 2020]todo-other-citations.

**Significance of proposed thesis** Probabilistic programming has been applied to a large number of problems in statistical inference; in fact, many of its original applications involved automating (with a convenient programming interface) traditional statistical inference tasks such as regression todo-cite and hierarchical modeling todo-cite. Although probabilistic graphical models such as factor graphs have been applied widely in robotics, with deep roots in the SLAM community todo-cite, these methods have experienced some of the same hurdles as traditional model-based verification methods discussed above: it is difficult to derive these models by hand, particularly for complex dynamical systems found in traditional "black-box" settings. This thesis will close this gap by applying probabilistic programming to automatically generating these models for robotics safety verification and optimization problems.

# 3 Local methods for design & verification

To design complex systems, engineers in many fields use computer-aided tools to boost their productivity. Mechanical engineers can use a suite of 3D CAD (computer-aided design) and FEA (finite-element analysis) tools to design structures and understand their performance. Likewise, electrical engineers use electronic design automation tools, including hardware description languages like Verilog, to design and analyze large-scale, reliable, and yet highly complex integrated circuits. Sadly, when it comes to designing autonomous systems and robots, engineers often take an ad-hoc approach, relying heavily on experience and tedious parameter tuning.

Two factors have made it difficult to develop automated design tools for robotics. The first is complexity: most robots are composed of many interacting subsystems. Although some tools may aid in designing certain subsystems (e.g. Simulink for controllers, SolidWorks or CATIA for hardware, custom software for training perception systems), these tools cover only a small part of the overall robotics design problem, which includes sensing, actuation, perception, navigation, control, and decision-making subsystems. In addition to being interconnected, these subsystems often have a large number of parameters that require tuning to achieve good performance (neural network-based perception is an extreme example of this trend). Moreover, since few robotic systems are exactly alike, an effective design tool must allow the user to select an appropriate level of abstraction for the problem at hand. As a result, there is a need for flexible computational tools that can help designers optimize complex robotic systems.

The second difficulty is uncertainty. Robots operate in dynamic environments that cannot be fully specified *a priori*, and nonlinear interactions between the robot and its environment can make this uncertainty difficult to quantify. Nevertheless, we must account for this uncertainty during the design process and ensure that our designs perform robustly. The nature of this uncertainty can vary from problem to problem, reiterating the requirement that an automated design tool must be flexible enough to adapt to different robot design problems.

To be successful, an automated robot design tool must address these two challenges (complexity and uncertainty). In addition, just as mechanical and electrical engineers use automated tools to both *design* and *verify* their designs, a robot design tool must enable its user to both design autonomous systems and test the robustness of those designs. In addition, we would also like the ability to close the loop between design and verification, feeding the results of verification back into the design process to make the system more robust (i.e. *verification-guided design*).

In this section, we frame these three tasks as optimization problems in order to develop local optimization-based solution methods. In doing so, we introduce terminology and notation that will be used in the rest of this proposal. Approaching these problems through the lens of optimization is a natural first step, and we will show empirically that it can lead to good results in a number of robotics settings; however, a purely optimization-based mindset has several limitations, which we will discuss at the end of this chapter with an eye towards resolving in the next.

## 3.1 Problem statement

At the heart of any design process is the tension between the factors a designer can control and those she cannot. For instance, a designer might be able to choose the locations of sensors and tune controller gains, but she cannot choose the sensor noise or disturbances (e.g. wind) encountered during operation. Robot design is therefore the process of choosing feasible values for the controllable factors (here referred to as *design parameters*) that achieve good performance despite the influence of uncontrollable factors (*exogenous parameters*).

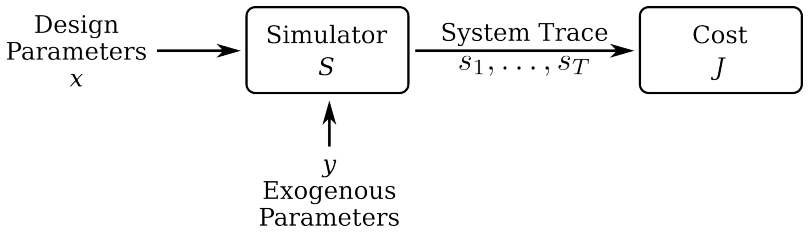



**Figure 3.1** | A high-level model of the design and verification problem for an autonomous system. Design optimization involves finding a set of design parameters so that the simulated cost is minimized, while verification involves understanding how changes in the exogenous parameters affect the simulated cost.

Of course, this is a deliberately narrow view of engineering design, since it focuses on parameter optimization and ignores important steps like problem formulation and system architecture selection. Our focus on parameter optimization is intentional, as it allows the designer to focus her creative abilities and engineering judgment on the architecture problem, using computational aids as interactive tools in a larger design process [Sharpe, 2021, Cascaval et al., 2021]. This focus is common in design optimization (e.g. aircraft design in [Sharpe, 2021] and 3D CAD optimization in [Cascaval et al., 2021]). Formally, we can model the design process as having four components, as illustrated in Fig. 3.1.

**Design parameters** The system designer has the ability to tune certain continuous free parameters $x \in \mathcal{X} \subseteq \mathbb{R}^n$; e.g., control gains or the positions of nodes in a sensor network. We assume that constraints on the design parameters are captured in the structure of $\mathcal{X}$.

**Exogenous parameters** Some factors are beyond the designer's control, such as wind speeds or sensor noise. We model these effects as continuous paramaters $y \in \mathcal{Y} \subseteq \mathbb{R}^m$, which we assume follow some prior likelihood $p_{0,y}$ supported on $\mathcal{Y}$. In this section, we assume no knowledge of $p_{0,y}$ other than the ability to sample from it, but we will revisit this assumption in subsequent chapters to see what insights can be gained from the structure of $p_{0,y}$.

**Simulator** Given particular choices for $x$ and $y$, the system's state $s \in \mathcal{S}$ evolves in discrete time according to a known simulator $S : \mathcal{X} \times \mathcal{Y} \mapsto \mathcal{S}^T$. This simulator describes the system's behavior over a finite horizon $T$ as a trace of states $s_1, \ldots, s_T$. We assume that $S$ is deterministic; randomness must be "imported" via the exogenous parameters.

**Cost** We assume access to a function $J : \mathcal{S}^T \mapsto \mathbb{R}$ mapping system behaviors (i.e. a trace of states) to a scalar performance metric that the designer seeks to minimize.

In this context, we can formalize the design, verification, and verification-guided design tasks as optimization problems. For the time being, we take a worst-case approach to verification, but we will revisit this in Section 4.

$$\textit{Design: } x = \arg\min_{x \in \mathcal{X}} \mathop{\mathbb{E}}_{y \sim \mathcal{Y}} [J(S(x, y))] \tag{3.1}$$

$$\textit{Verification: } y = \arg\min_{y \in \mathcal{Y}} J(S(x, y)) \tag{3.2}$$

$$\textit{Verification-guided design: } x = \arg\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} J(S(x, y)) \tag{3.3}$$

These problems have been studied in various forms throughout the prior literature, but previous works tend to either use sample-inneficient black-box optimization techniques [Corso et al., 2019, Corso et al., 2021, Ding et al., 2020, Wang et al., 2021] or focus on developing application-specific algorithms for certain cases where gradients can be derived [Schulz et al., 2017, Du et al., 2016, Du et al., 2021, Ma et al., 2021, Xu et al., 2019]. The first contribution in this chapter is to expand on these application-specific works to demonstrate how automatic differentiation enables us to easily apply gradient-based optimization methods to solve the design and verification problems.

Unfortunately, while simply feeding AD-derived gradients into off-the-shelf optimizers is a natural (and often effective) first step, it provides few assurances of that our optimized designs will be robust, and does little to connect the results of verification back into the design process. The second contribution in this chapter is to close this gap by developing a framework that uses AD-derived gradients to accelerate adversarial testing.

This chapter presents work published in RSS 2022 [Dawson and Fan, 2022a] and IROS 2022 [Dawson and Fan, 2022b]. In the interest of brevity, we defer implementation details to the appendix and focus here on the key ideas, results, and limitations.

## 3.2 Efficient design optimization using differentiable simulation

A natural first instinct when presented with gradients from an automatic differentiation system is to apply a gradient-based optimizer to find low-cost design parameters. This approach is simple (relying on pre-existing optimization methods), flexible (using general-purpose AD libraries to model subsystems with varying levels of abstraction), and efficient (making use of gradients to accelerate the search for high-performing designs).

Although it is possible to directly optimize the objective in (3.1) to find low-cost design parameters, in many practical use-cases we are interested in designs that are not only low-cost but robust to variation in $y$, which we can encode by regularizing the objective with respect to the variance of the cost:

$$\min_{x \in \mathcal{X}} \quad \mathop{\mathbb{E}}_{y \sim \mathcal{Y}} \Big[ J \circ S \left( x, y \right) \Big] + \lambda \mathrm{Var}_{\mathrm{y} \sim \mathcal{Y}} \Big[ \mathrm{J} \circ \mathrm{S} \left( \mathrm{x}, \mathrm{y} \right) \Big] \tag{3.4a}$$

To yield a tractable problem, we replace the expectation and variance with unbiased estimates over $N$ samples $y_i \sim \mathcal{Y}, i = 1, \ldots, N$.

$$\min_{x \in \mathcal{X}} \quad \frac{1}{N} \sum_{i=1}^{N} \Big[ J \circ S \left( x, y_i \right) \Big] + \lambda \left[ \frac{\sum_{i=1}^{N} \left( J \circ S \left( x, y_i \right) \right)^2}{N - 1} - \frac{\left( \sum_{i=1}^{N} J \circ S \left( x, y_i \right) \right)^2}{(N-1)N} \right] \tag{3.5a}$$

Since computing this objective requires multiple calls to the simulator $S$, approximating the gradients of (3.5a) using finite-differences or stochastic multi-sample estimators would incur a

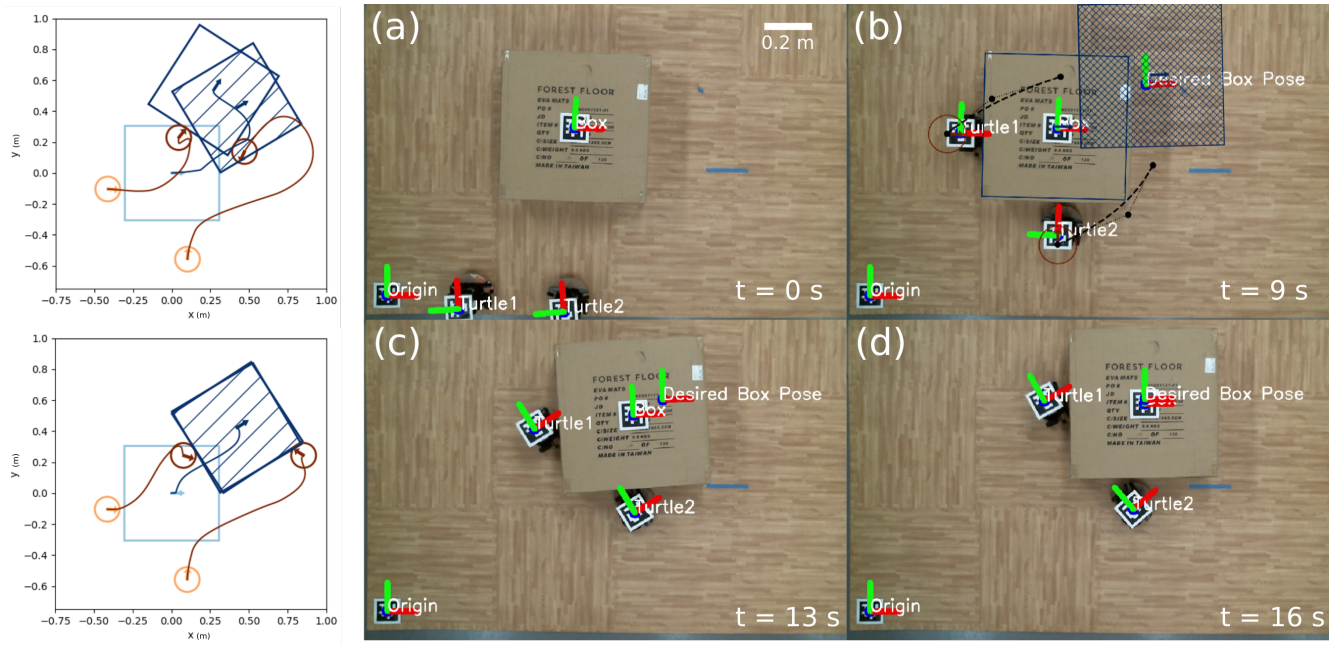



**Figure 3.2 |** Left: Initial (top) and optimized (bottom) manipulation strategies in simulation (light/dark colors indicate initial/final positions, stripes indicate desired position). Right: Optimized manipulation strategy deployed in hardware (video included in the supplementary materials). (a) The robots first move to positions around the box. (b) Using the optimized neural network, the robots plan a cubic spline trajectory pushing the box to its desired location. (c-d) The robots execute the plan by tracking that trajectory.

large computational cost (requiring $O(N \dim x)$ calls to $S$). Fortunately, backwards-mode AD allows us to obtain these gradients much more cheaply, with a constant $O(N)$ simulator calls, regardless of the dimension of the search space. On high-dimensional problems, like optimizing a neural network planning system for the multi-robot collaborative manipulation system shown in Fig. 3.2, the improved scaling of AD is clearly apparent. In this example, where the cost measures how close the robots were able to maneuver the box to its desired pose (which varies randomly along with uncertain coefficients of friction in the environment), $\dim x = 454$ and $N = 512$, so AD requires two orders of magnitude fewer simulator evaluations than a finite-difference scheme.

This improved scaling can be seen empirically if we compare the time required to run L-BFSG-B (a quasi-Newton first-order optimizer) until convergence using both AD-derived and finite-difference gradients. Fig. 3.3 shows that although both methods achieve a similar optimal cost, AD-based optimization runs 20x faster (even after accounting for the additional computational overhead of backwards-mode AD).

## 3.3 Local adversarial testing for verification-guided design

The results in the previous section show how we can use AD for efficient and robust gradient-based optimization; however, the computational expense of averaging over exogenous parameters makes it difficult to scale to larger systems that may be more difficult to simulate. In addition, in many safety-critical contexts we are interested in verifying not only the average-case but the worst-case behavior.

These difficulties motivate the next contribution of this chapter, where we introduce an adversarial optimization approach to solving the min/max problem (3.3). Of course, solving (3.3) to global optimality in the general nonlinear case is intractable. Instead, we take advantage of

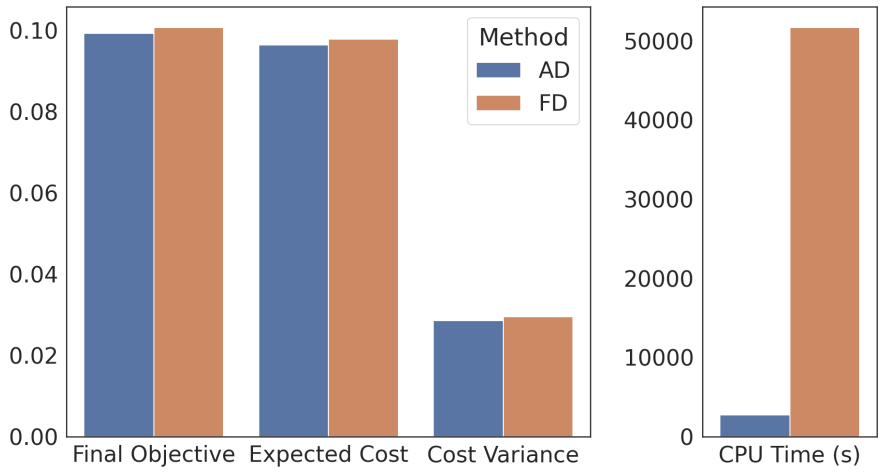



**Figure 3.3** | Improvement of automatic differentiation (AD) over finite differences (FD) on the multi-agent manipulation case study shown in Fig. 3.2.

this game structure to design an iterative algorithm to find the *generalized Nash equilibrium*: the design parameters $x$ and corresponding $y$ such that neither the planner nor the adversary have an incentive to change their choice [Facchinei and Kanzow, 2007].

To solve for such an equilibrium, a common strategy is the family of nonlinear Gauss-Seidel-type methods. These methods solve max-min problems like (3.3) by alternating between $x$ and $y$, tuning one set of parameters while keeping the others constant; i.e. alternating between the two optimization problems:

$$x^* = \arg\min_x J(x, y^*) \tag{3.6a}$$

$$y^* = \arg\max_y J(x^*, y) \tag{3.6b}$$

Although these methods are not guaranteed to converge, it is known that if they do, then the convergence point $(x^*, y^*)$ is a Nash equilibrium [Facchinei and Kanzow, 2007].

A risk of applying such a simple alternating scheme is that the nonlinear optimization for both $x$ and $y$ can easily get caught in local minima, which increases the risk of "overfitting" the design to a particular value of $y^*$. To mitigate the risk of overfitting and improve the robustness of our optimized design, we extend the standard Gauss-Seidel method with two ideas from the machine learning and optimization literature. First, we take inspiration from the success of domain randomization in robust machine learning [Tobin et al., 2017]: instead of optimizing $x$ with respect to a single fixed $y^*$, we can maintain a dataset $\mathcal{Y}_N = \{y_i\}_{i=1,\dots,N}$ and optimize the performance of $x$ across all of these samples:

$$x^* = \arg\min_x \mathbb{E}_{\mathcal{Y}_N}\left[J(x, y_i)\right] \tag{3.7a}$$

$$y^* = \arg\max_y J(x^*, y) \tag{3.7b}$$

This domain-randomized objective is similar to that used in the previous section and has the potential to improve the robustness of the resulting equilibria, but it is relatively sample inefficient; it may require a large number of random samples $y_i$ to cover the range of possible system behaviors.

To address this sample inefficiency, we take inspiration from a second idea in the optimization and learning literature: learning from counterexamples [Chang et al., 2019]. The key insight here is that we can do better than simply randomly sampling $y_i$; we can use the values of $y^*$ found during successive iterations of the Gauss-Seidel process as high-quality counterexamples to guide the optimization of $x$. This insight results in our counterexample-guided Gauss-Seidel optimization method, which is outlined in pseudocode in Algorithm 3.1.

Our algorithm proceeds as follows. We begin by initializing the dataset with $N_0$ i.i.d. examples $y_i$, then we alternate between solving the two optimization problems in (3.7a). At each iteration, we add our current estimate of the adversary's best response $y^*$ to the dataset, and we stop either when the algorithm reaches a fixed point (the adversary's best response after solving (3.7a) is the same as the best response from the previous round) or when a maximum number of iterations is reached. As we will show in our experiments in the next section, this counterexample-guided optimization achieves a higher sample efficiency than simple domain randomization — it finds designs that are more robust to adversarial disturbance while considering a much smaller dataset. Although our use of nonlinear optimization means that our algorithm is not complete, we find empirically that it succeeds in finding a satisfactory design in the large majority of cases.

It is important to note that this algorithm is enabled by automatic differentiation; without access to the gradients of $J$ it would be much more difficult to solve the subproblems in lines 4 and 5 of Algorithm 3.1. We are not aware of any approaches that make use of automatically-derived gradients with respect to disturbance parameters for adversarial verification-guided optimization.

**Algorithm 3.1:** Counterexample-guided Gauss-Seidel method for solving robust planning problems

**Input:** Starting dataset size $N_0$
Maximum number of iterations $M$
**Output:** Optimized design parameters $x^*$
Dataset of counterexamples $\{y^*\}$

1 $\{y^*\} \leftarrow N_0$ examples $y_i \in \mathcal{Y}$ sampled uniformly i.i.d.
2 $y^*_{prev} \leftarrow \varnothing$
3 **for** $i \in \{1, \dots, M\}$ **do**
4  $x^* = \arg\min_x \mathbb{E}_{\mathcal{Y}_N}[J(x, y_i)]$
5  $y^* = \arg\max_y J(x^*, y)$
6  **if** $y^* = y^*_{prev}$ **then**
7  **break**
8  $y^*_{prev} \leftarrow y^*$
9  Append $y^*$ to $\{y^*\}$
10 **return** $x^*$, $\{y^*\}$

To validate this approach, we can use two case studies involving planning and control for the satellite rendezvous problem posed in [Jewison and Erwin, 2016]. We define this planning problem using signal temporal logic (STL), which provide a formal language for capturing complex behavioral requirements and specifying how a robot behaves over time. For a brief introduction to the syntax and semantics of STL, see [Dawson and Fan, 2022b]. By restricting this case study to focus on planning, we are able to benchmark against state-of-the-art planning algorithms to show the robustness and scalability benefits of our adversarial optimization-based approach. We implement

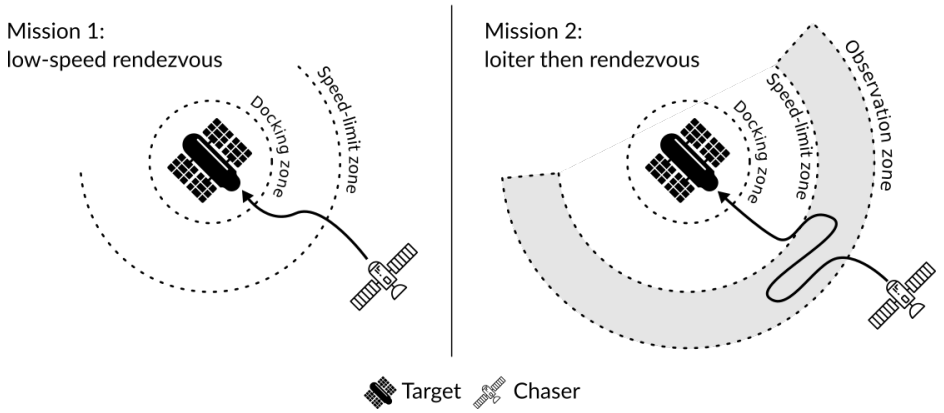



**Figure 3.4** | Two satellite rendezvous missions used to test our adversarial optimization framework. In the first mission, the chaser satellite must eventually reach the target while respecting a maximum speed constraint in the region immediately around the target. In the second mission, the chaser must still reach the target and obey the speed limit, but it must also loiter in an observation region for some minimum time before approaching. The first mission requires an STL formula with three predicates and three temporal operators, while the second mission requires five predicates and five temporal operators.

a smooth differentiable version of STL following the approach in [Leung et al., 2021, Pant et al., 2017].

In this satellite rendezvous problem, the goal is to maneuver a chaser satellite to catch a target satellite. In this setting, we construct STL specifications for two rendezvous missions: a simple low-speed rendezvous and a more complex loiter-then-rendezvous mission, illustrated in Fig. 3.4. The STL specifications for each mission, $\psi_1$ and $\psi_2$, are given formally as:

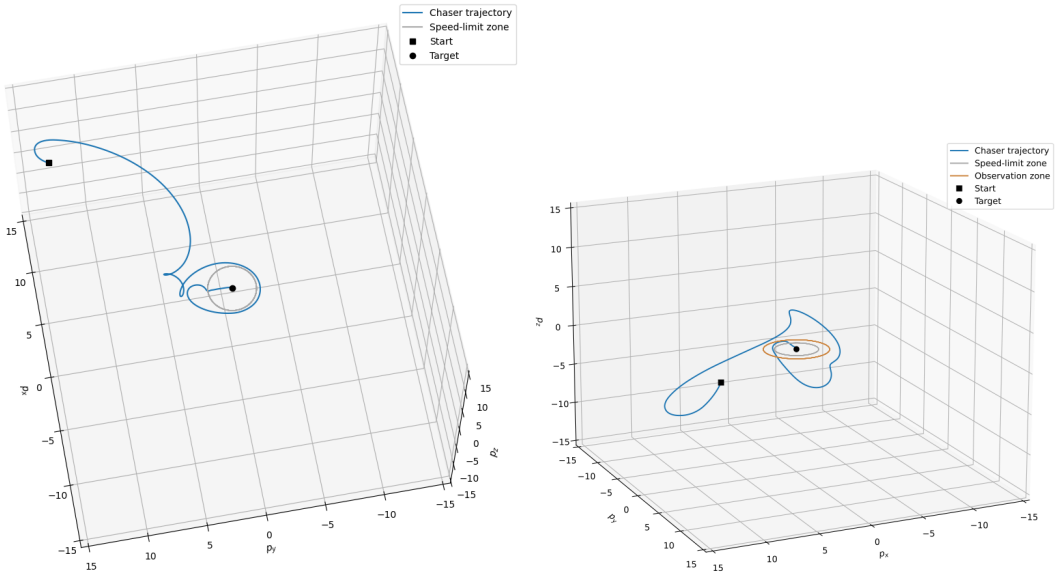
$$\begin{aligned}
\psi_1 &= \psi_{\text{reach target}} \wedge \psi_{\text{speed limit}} \\
\psi_2 &= \psi_{\text{reach target}} \wedge \psi_{\text{speed limit}} \wedge \psi_{\text{loiter}} \\
\psi_{\text{reach target}} &= \Diamond\,(r \leq 0.1) \\
\psi_{\text{speed limit}} &= (r \geq 2.0)\,\mathcal{U}\,\Box\,(v \leq 0.1) \\
\psi_{\text{loiter}} &= \Diamond\Box_{[0,T_{obs}]}\,(2.0 \leq r \wedge r \leq 3.0)
\end{aligned}$$

where $r = \sqrt{p_x^2 + p_y^2 + p_z^2}$ and $v = \sqrt{v_x^2 + v_y^2 + v_z^2}$.

These formulae can be read as follows: $\psi_{\text{reach target}}$ specifies that "the chaser eventually comes within 0.1 m of the target", $\psi_{\text{speed limit}}$ specifies that "once the chaser is within 2.0 m of the target, its speed cannot exceed 0.1 m/s", and $\psi_{\text{loiter}}$ specifies that "at some point during the mission, the chaser should spend $T_{obs}$ seconds between 2–3 m away from the target". The two missions are build from these three building blocks: mission 1 includes the "reach target" and "speed limit" requirement, while mission 2 includes all three requirements.

For each mission, the design parameters $x$ include both state/input waypoints along a planned trajectory and the feedback gains used to track that trajectory, and the exogenous parameters $y$ represent bounded uncertainty in the initial state of the chaser ($p_x(0), p_y(0) \in [10, 13]$, $p_z(0) \in [-3, 3]$, $v_x(0), v_y(0), v_z(0) \in [-1, 1]$). We use a 200 s-long simulation with a 2 s timestep for both missions, and $T_{obs} = 10$ s.

For each mission $i = 1, 2$, we define a cost function as $J_i = -\rho_i + \lambda I$, where $\rho_i = \rho(\psi_i, S(x, y), 0)$ is the STL robustness margin at the start of the trajectory (i.e. how satisfied/unsatisfied the requirements are), $I$ is the total impulse required to execute the maneuver (in Newton-seconds),




**(a)** Mission 1 **(b)** Mission 2

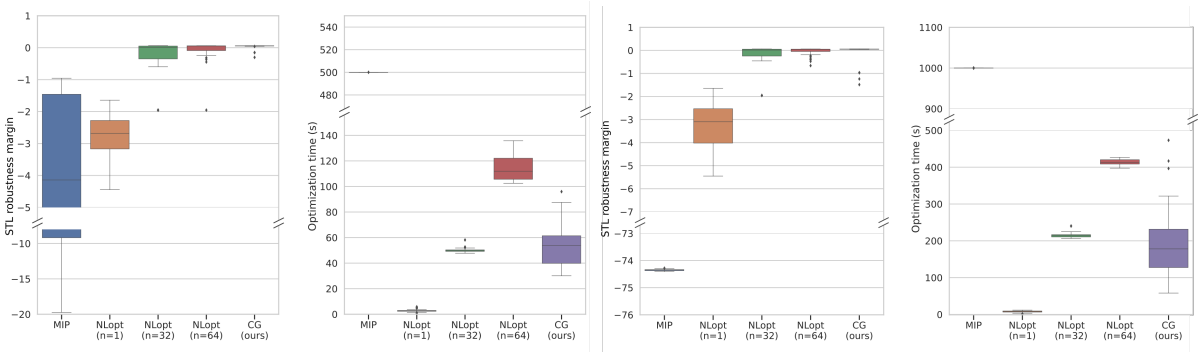
**Figure 3.5 |** (a) The optimized trajectory found using our counterexample-guided optimization strategy for mission 1 (rendezvous with speed constraint). The chaser satellite only enters the speed-limit zone once it has slowed down sufficiently. (b) The optimized trajectory for mission 2 (loiter then rendezvous with speed constraint), satisfying the additional mission requirement of spending time in the observation region before approaching the target.

and $\lambda = 5 \times 10^{-5}$. By applying our iterative counterexample-guided optimization strategy to this problem, we find the optimized trajectories for mission 1 and 2 shown in Fig. 3.5 along with the worst-case $y$. In these examples, we use $N_0 = 8$ initial examples and $M = 10$ maximum rounds, but the algorithm converges in less than 10 rounds in all trials. In both missions, our approach reliably finds a solution that remains feasible despite worst-case variation in the exogenous parameters, achieving a positive STL robustness margin in $> 90\%$ of trials in each case. Our counterexample-guided approach requires an average of 53.7 s to solve mission 1 and 194.2 s to solve mission 2 (averaged across 50 trials).

To quantify the advantage of our AD-enabled adversarial optimization method relative to the state-of-the-art, there are two salient comparisons. First, what advantage does adversarial optimization have relative to domain randomization? Second, since we are solving a specialized motion planning problem, how does our method compare against application-specific solvers for this STL planning problem?

To answer the first question, we compare our adversarial method with non-adversarial optimization with domain randomization with 64 and 32 samples, as well as non-adversarial optimization without any domain randomization (1 sample). To answer the second question, we compare with a mixed-integer programming (MIP) approach to solving STL planning problems [Sun et al., 2022]. The results of this comparison are shown in 3.6. All experiments were run on a laptop computer with 8 GB RAM and a 1.8 GHz 8-core processor, with no GPU, and results are averaged over 50 random seeds.

We find that our method is consistently more robust than prior methods; in the first mission, it satisfies the STL specification in all but 3 trials, despite adversarial disturbances. For comparison, the next-best method (domain randomization with 64 samples) failed to solve the first mission in 14 out of 50 trials and took more than twice as long on average to find a plan (114.3 s as opposed to 53.7 s for our method). This advantage is due to the quality of the examples used




**(a)** Mission 1 **(b)** Mission 2

**Figure 3.6 |** Comparison of different STL planning methods on mission 1 (a) and 2 (b), averaged over 50 random seeds. (left subfigures) The robustness margin $\rho(\psi_1)$ computed for the optimized design parameters and worst-case exogenous parameters. (right subfigures) The planning time required by each method. Our method (CG) achieves much higher robustness than all other methods (satisfying the STL specification despite adversarial perturbations in all but 3 instances) and runs twice as fast as the next-most-robust method.

during optimization; instead of 64 random samples, our method uses 8 initial random samples and between 1 and 4 counterexamples (median 2) representing worst-case variation in $y$, making our method much more sample-efficient. This pattern also holds on the second mission, where 64-sample domain randomization takes more than twice as long as our method and fails to satisfy the STL specification in 17 out of 50 trials (compared to only 4 failures for our method). Our method required a median of 2 counterexamples in addition to the 8 initial examples to solve the second mission (the slowest trial required 7 additional examples).

We also find that our method finds more robust solutions than the MIP method, since MIP cannot tractably consider variation in $y$ (the MIP method is also unable to find a feasible solution within 500 s in 16 out of 50 trials). MIP's performance also suffers due to discretization error, since we were forced to discretize the continuous-time dynamics with relatively few knot points (one every 2 s) to yield a tractable MIP optimization problem.

## 3.4 Discussion & Limitations

In this chapter, I have presented my previous work on auto-diff-enabled design optimization and verification, with three key takeaways. The first two takeaways are technical and regard the utility of AD for design optimization and verification. First: AD enables a flexible, general-purpose framework for optimizing and verifying a wide-range of robotics problems, with support for multiple interacting subsystems, rich dynamics, and complex task specifications. Second: the use of gradients from AD enables solution methods (such as variance regularization and adversarial optimization) that would be much more computationally expensive if these gradients had to be estimated by other means. The final takeaway is agnostic to the use of AD and concerns how using adversarial optimization can improve both the robustness and the sample efficiency of design optimization (relative to domain randomization).

The main limitation of the work presented in this chapter is its reliance on local gradient-based optimization. As a result, there is still a need for algorithms that take a more global perspective, particularly for verification, which we will introduce in the next chapter.

# 4 Global methods for design and verification

## 4.1 From optimization to inference

### 4.1.1 Gradient-accelerated automated inference

## 4.2 Sampling diverse failure modes

### 4.2.1 Case study: predicting transmission outages in electrical power networks

## 4.3 Repairing failure modes

### 4.3.1 Case study: robust generation dispatch for secure power networks

# 5 Future work

## 5.1 Pushing the limits of differentiability: vision, manipulation, and beyond

## 5.2 Theoretical analysis

## 5.3 Probabilistic programming

## 5.4 Static program analysis

# 6 Milestones and Program Logistics

## 6.1 Classes and Degree Milestones

Table 6.1 shows my competed coursework, and Table 6.2 shows completed and anticipated degree milestones.

**Table 6.1 |** My completed coursework, satisfying all academic requirements for the doctoral program. Major: autonomy. Minor: controls.

| Semester | Class | Req. | Status |
|---|---|---|---|
| Fall 2019 | 16.413 Principles of Autonomy & Decision Making | major | completed |
| Fall 2019 | 6.255 Optimization Methods | major/math | completed |
| Spring 2020 | 16.412 Cognitive Robotics | major | completed |
| Spring 2020 | 6.832 Underactuated Robotics | minor | completed |
| Fall 2020 | 18.385 Nonlinear Dynamics and Chaos | minor/math | completed |
| Fall 2020 | 2.160 Identification, Estimation, and Learning | minor | completed |
| Spring 2021 | 16.S398 Formal Methods in Autonomy | major | completed |
| Fall 2021 | 6.843: Robotic Manipulation | major | completed |
| Fall 2021 | 16.995 Doctoral Research & Communication Seminar | RPC | completed |

**Table 6.2 |** Milestones towards my completion of the doctoral degree. Italicized milestones are anticipated.

| | |
|---|---|
| Fall 2019 (September) | Began studies at MIT |
| Fall 2020 (December) | Field evaluation complete |
| Spring 2021 (May) | Masters thesis submitted |
| Fall 2022 (September) | Committee meeting #1 |
| Summer 2023 (June) | Commitee meeting #2 and thesis proposal |
| *Fall 2023* | *Committee meeting #3* |
| *Spring 2024* | *Committee meeting #4 and green light* |
| *Spring/Summer 2024* | *Thesis defense* |

## 6.2 Research Schedule

My thesis research will proceed in stages, as outlined below.

Spring 2022

1. Certifiable robot design optimization using differentiable programming
   (a) Develop design optimization tool using automatic differentiation
   (b) Develop statistical robustness certification tool based on extremal types theorem
   (c) Hardware deployment
   (d) Accepted to RSS 2022
2. Robust counterexample-guided optimization with temporal logic specifications
   (a) Define two-player zero-sum game between the designer and the verifier

(b) Incorporate counterexamples from the verifier to guide robust design optimization

(c) Use differentiable signal temporal logic for complex task specification

(d) Submitted to IROS 2022

Fall/Winter 2022

1. Improving design optimization and verification through automated failure mode discovery

   (a) Use differentiable programming and sequential MCMC to discover a diverse set of possible failure modes.

   (b) Use counterexamples from all failure modes to guide robust design optimization

Spring 2023

1. Extend failure mode prediction and mitigation framework to challenging new problem domains, including:

   (a) Perception-in-the-loop controllers, through the use of differentiable rendering,

   (b) Deep neural network controllers.

*Fall 2023*

1. Develop MCMC-based algorithm for safety verification and optimization of systems with discrete structure (e.g. involutive MCMC for robot manipulators with variaible morphologies).

2. Write thesis

*Spring 2024*

1. Write thesis

2. Defend thesis and graduate

# References


[Agrawal et al., 2019] Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S., and Kolter, J. Z. (2019). Differentiable Convex Optimization Layers. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

[Ahmadi and Majumdar, 2016] Ahmadi, A. A. and Majumdar, A. (2016). Some applications of polynomial optimization in operations research and real-time decision making. *Optimization Letters*, 10(4):709–729.

[Amodei et al., 2016] Amodei, D., Olah, C., Steinhardt, J., Christiano, P. F., Schulman, J., and Mané, D. (2016). Concrete problems in AI safety. *ArXiv*, abs/1606.06565.

[Amos and Kolter, 2017] Amos, B. and Kolter, J. Z. (2017). OptNet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 136–145, Sydney, NSW, Australia. JMLR.org.

[Annpureddy et al., 2011a] Annpureddy, Y., Liu, C., Fainekos, G., and Sankaranarayanan, S. (2011a). S-TaLiRo: A tool for temporal logic falsification for hybrid systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6605 LNCS:254–257.

[Annpureddy et al., 2011b] Annpureddy, Y., Liu, C., Fainekos, G., and Sankaranarayanan, S. (2011b). S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6605 LNCS:254–257.

[Bansal et al., 2017] Bansal, S., Chen, M., Herbert, S., and Tomlin, C. J. (2017). Hamilton-Jacobi reachability: A brief overview and recent advances. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2242–2253.

[Belta and Sadraddini, 2019] Belta, C. and Sadraddini, S. (2019). Formal Methods for Control Synthesis: An Optimization Perspective. *Annual Review of Control, Robotics, and Autonomous Systems*, 2(1):115–140.

[Benard et al., 2000] Benard, D., Dorais, G. A., Gamble, E., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Nayak, P., Rouquette, N., Rajan, K., and Norvig, P. (2000). Remote Agent Experiment.

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

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym.

[Cascaval et al., 2021] Cascaval, D., Shalah, M., Quinn, P., Bodik, R., Agrawala, M., and Schulz, A. (2021). Differentiable 3D CAD programs for bidirectional editing. *arXiv*, abs/2110.01182.

[Chang et al., 2019] Chang, Y.-C., Roohi, N., and Gao, S. (2019). Neural Lyapunov Control. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

[Chou et al., 2018] Chou, G., Sahin, Y. E., Yang, L., Rutledge, K. J., Nilsson, P., and Ozay, N. (2018). Using control synthesis to generate corner cases: A case study on autonomous driving. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2906–2917.

[Corso et al., 2019] Corso, A., Du, P., Driggs-Campbell, K., and Kochenderfer, M. J. (2019). Adaptive Stress Testing with Reward Augmentation for Autonomous Vehicle Validatio. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 163–168.

[Corso and Kochenderfer, 2020] Corso, A. and Kochenderfer, M. J. (2020). Interpretable Safety Validation for Autonomous Vehicles. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6.

[Corso et al., 2021] Corso, A., Moss, R., Koren, M., Lee, R., and Kochenderfer, M. (2021). A Survey of Algorithms for Black-Box Safety Validation of Cyber-Physical Systems. *Journal of Artificial Intelligence Research*, 72:377–428.

[Cusumano-Towner et al., 2020] Cusumano-Towner, M., Lew, A. K., and Mansinghka, V. K. (2020). Automating Involutive MCMC using Probabilistic and Differentiable Programming.

[Dawson and Fan, 2022a] Dawson, C. and Fan, C. (2022a). Certifiable Robot Design Optimization using Differentiable Programming. In *Robotics: Science and Systems XVIII*, volume 18.

[Dawson and Fan, 2022b] Dawson, C. and Fan, C. (2022b). Robust Counterexample-guided Optimization for Planning from Differentiable Temporal Logic. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7205–7212.

[de Avila Belbute-Peres et al., 2018] de Avila Belbute-Peres, F., Smith, K., Allen, K., Tenenbaum, J., and Kolter, J. Z. (2018). End-to-end differentiable physics for learning and control. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

[de Kleer and Williams, 1987] de Kleer, J. and Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130.

[Ding et al., 2020] Ding, W., Chen, B., Xu, M., and Zhao, D. (2020). Learning to Collide: An Adaptive Safety-Critical Scenarios Generating Method. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2243–2250.

[Du et al., 2021] Du, T., Hughes, J., Wah, S., Matusik, W., and Rus, D. (2021). Underwater soft robot modeling and control with differentiable simulation. *IEEE Robotics and Automation Letters*.

[Du et al., 2016] Du, T., Schulz, A., Zhu, B., Bickel, B., and Matusik, W. (2016). Computational multicopter design. *ACM Transactions on Graphics (TOG)*, 35(6):227.

[Facchinei and Kanzow, 2007] Facchinei, F. and Kanzow, C. (2007). Generalized Nash equilibrium problems. *4OR*, 5(3):173–210.

[Hanselmann et al., 2022] Hanselmann, N., Renz, K., Chitta, K., Bhattacharyya, A., and Geiger, A. (2022). KING: Generating Safety-Critical Driving Scenarios for Robust Imitation via Kinematics Gradients.

[Howell et al., 2022] Howell, T., Le Cleac'h, S., Kolter, Z., Schwager, M., and Manchester, Z. (2022). Dojo: A differentiable simulator for robotics. *arXiv preprint arXiv:2203.00806*.

[Hu et al., 2019] Hu, Y., Anderson, L., Li, T.-M., Sun, Q., Carr, N., Ragan-Kelley, J., and Durand, F. (2019). DiffTaichi: Differentiable Programming for Physical Simulation. In *International Conference on Learning Representations*.

[Jewison and Erwin, 2016] Jewison, C. and Erwin, R. S. (2016). A spacecraft benchmark problem for hybrid control and estimation. *2016 IEEE 55th Conference on Decision and Control, CDC 2016*, pages 3300–3305.

[Kochenderfer and Wheeler, 2019] Kochenderfer, M. J. and Wheeler, T. A. (2019). *Algorithms for Optimization*. The MIT Press, Cambridge, Massachusetts.

[Kress-Gazit et al., 2018] Kress-Gazit, H., Lahijanian, M., and Raman, V. (2018). Synthesis for Robots: Guarantees and Feedback for Robot Behavior. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1):211–236.

[Le Lidec et al., 2021] Le Lidec, Q., Laptev, I., Schmid, C., and Carpentier, J. (2021). Differentiable rendering with perturbed optimizers. In *Advances in Neural Information Processing Systems*, volume 34, pages 20398–20409. Curran Associates, Inc.

[Leung et al., 2021] Leung, K., Arechiga, N., and Pavone, M. (2021). Back-Propagation Through Signal Temporal Logic Specifications: Infusing Logical Structure into Gradient-Based Methods. In LaValle, S. M., Lin, M., Ojala, T., Shell, D., and Yu, J., editors, *Algorithmic Foundations of Robotics XIV*, Springer Proceedings in Advanced Robotics, pages 432–449, Cham. Springer International Publishing.

[Ma et al., 2021] Ma, P., Du, T., Zhang, J. Z., Wu, K., Spielberg, A., Katzschmann, R. K., and Matusik, W. (2021). DiffAqua: A differentiable computational design pipeline for soft underwater swimmers with shape interpolation. *ACM Transactions on Graphics (TOG)*, 40(4):132.

[Ma et al., 2019] Ma, Y. A., Chen, Y., Jin, C., Flammarion, N., and Jordan, M. I. (2019). Sampling can be faster than optimization. *Proceedings of the National Academy of Sciences of the United States of America*, 116(42):20881–20885.

[Majumdar et al., 2014] Majumdar, A., Ahmadi, A. A., and Tedrake, R. (2014). Control and verification of high-dimensional systems with DSOS and SDSOS programming. In *53rd IEEE Conference on Decision and Control*, pages 394–401.

[Metz et al., 2022] Metz, L., Freeman, C. D., Schoenholz, S. S., and Kachman, T. (2022). Gradients are Not All You Need.

[O' Kelly et al., 2018] O' Kelly, M., Sinha, A., Namkoong, H., Tedrake, R., and Duchi, J. C. (2018). Scalable End-to-End Autonomous Vehicle Testing via Rare-event Simulation. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

[Olson, 2011] Olson, E. (2011). AprilTag: A robust and flexible visual fiducial system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3400–3407. IEEE.

[Pant et al., 2017] Pant, Y. V., Abbas, H., and Mangharam, R. (2017). Smooth Operator: Control using the Smooth Robustness of Temporal Logic. *IEEE Conference on Control Technology and Applications, 2017*.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library.

[Posada, ] Posada, S. EA 22-002: Autopilot & First Responder Scenes. *US DOT NHTSA*.

[Riedmaier et al., 2020] Riedmaier, S., Ponn, T., Ludwig, D., Schick, B., and Diermeyer, F. (2020). Survey on Scenario-Based Safety Assessment of Automated Vehicles. *IEEE Access*, 8:87456–87477.

[Schulz et al., 2017] Schulz, A., Sung, C., Spielberg, A., Zhao, W., Cheng, R., Grinspun, E., Rus, D., and Matusik, W. (2017). Interactive robogami: An end-to-end system for design of robots with ground locomotion. *The International Journal of Robotics Research*, 36(10):1131–1147.

[Sharpe, 2021] Sharpe, P. D. (2021). *AeroSandbox: A Differentiable Framework for Aircraft Design Optimization*. PhD thesis, MIT.

[Suh et al., 2021] Suh, H., Pang, T., and Tedrake, R. (2021). Bundled gradients through contact via randomized smoothing. *ArXiv*, abs/2109.05143.

[Suh et al., 2022] Suh, H. J. T., Simchowitz, M., Zhang, K., and Tedrake, R. (2022). Do Differentiable Simulators Give Better Policy Gradients?

[Sun et al., 2022] Sun, D., Chen, J., Mitra, S., and Fan, C. (2022). Multi-agent Motion Planning from Signal Temporal Logic Specifications. *IEEE Robotics and Automation Letters (RA-L)*.

[Sun et al., 2021] Sun, H., Feng, S., Yan, X., and Liu, H. X. (2021). Corner Case Generation and Analysis for Safety Assessment of Autonomous Vehicles. *Transportation Research Record*, 2675(11):587–600.

[Tobin et al., 2017] Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30.

[Wang et al., 2021] Wang, J., Pun, A., Tu, J., Manivasagam, S., Sadat, A., Casas, S., Ren, M., and Urtasun, R. (2021). AdvSim: Generating Safety-Critical Scenarios for Self-Driving Vehicles. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9909–9918.

[Xu et al., 2022a] Xu, C., Ding, W., Lyu, W., Liu, Z., Wang, S., He, Y., Hu, H., Zhao, D., and Li, B. (2022a). SafeBench: A Benchmarking Platform for Safety Evaluation of Autonomous Vehicles. *Advances in Neural Information Processing Systems*, 35:25667–25682.

[Xu et al., 2019] Xu, J., Du, T., Foshey, M., Li, B., Zhu, B., Schulz, A., and Matusik, W. (2019). Learning to fly: Computational controller design for hybrid UAVs with reinforcement learning. *ACM Trans. Graph.*, 38(4).

[Xu et al., 2022b] Xu, J., Makoviychuk, V., Narang, Y., Ramos, F., Matusik, W., Garg, A., and Macklin, M. (2022b). Accelerated Policy Learning with Parallel Differentiable Simulation. In *International Conference on Learning Representations*.

[Zhang et al., 2022] Zhang, Q., Hu, S., Sun, J., Chen, Q. A., and Mao, Z. M. (2022). On Adversarial Robustness of Trajectory Prediction for Autonomous Vehicles. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15159–15168.

[Zhong et al., 2022] Zhong, Z., Rempe, D., Xu, D., Chen, Y., Veer, S., Che, T., Ray, B., and Pavone, M. (2022). Guided Conditional Diffusion for Controllable Traffic Simulation.

[Zhou et al., 2021] Zhou, Y., Booth, S., Figueroa, N., and Shah, J. (2021). RoCUS: Robot Controller Understanding via Sampling. In *5th Annual Conference on Robot Learning*.

# A Appendix: Details on local methods

## A.1 AD-enabled design optimization

To develop a general-purpose, efficient, and robust design optimization framework for autonomous systems, we must address three main challenges. First, how can we ensure that the system is *flexible*, not requiring large amounts of work to specialize the framework to particular application domains. Second, how can we incorporate *robustness* into the design optimization process, to minimize the sensitivity of the optimized design to variation in the exogenous parameters. Finally, how can we achieve both of these goals *efficiently*, making use of gradients wherever possible to accelerate the search for high-performing designs.

**Flexibility** Most robots are composed of many interacting subsystems, and different subsystems are typically modeled using different levels of abstraction. Although some tools may aid in designing certain subsystems (e.g. Simulink for controllers, SolidWorks for hardware), these tools cover only a small part of the overall robotics design problem, which includes sensing, actuation, perception, navigation, control, and decision-making subsystems. Since few robotic systems are exactly alike, an effective design tool must allow the user to select an appropriate level of abstraction for the problem at hand.

When it comes to managing complexity in a general-purpose design framework, programming languages are a natural tool. They allow users (i.e. programmers) to define precisely which abstractions are appropriate for any given application (e.g. by defining appropriate class hierarchies and function interfaces) without sacrificing generality. To take advantage of this expressivity, we can view engineering designs as programs that define the behavior of the system given suitable choices for design structure and parameters. We can then use automatic differentiation to derive gradients connecting these parameters to the system's behavior and optimize accordingly. This view is inspired by recent work in 3D design optimization [Cascaval et al., 2021], aircraft design [Sharpe, 2021], and machine learning [Paszke et al., 2019, Bradbury et al., 2018].

**Robustness** Robots operate in dynamic environments that cannot be fully specified *a priori*, and nonlinear interactions between the robot and its environment can make this uncertainty difficult to quantify. Nevertheless, we must account for this uncertainty during the design process and ensure that our designs perform robustly.

Due to this uncertainty, simply minimizing the expected value of the cost $\mathbb{E}_{y\sim\mathcal{Y}}\left[J\left(S\left(x,y\right)\right)\right]$ can lead to myopic behavior where exceptional performance for some values of $y$ compensates for poor performance on other values; this is related to the phenomenon of "reward hacking" in reinforcement learning [Amodei et al., 2016]. Ideally, we would like our designs to be robust to variations in exogenous parameters: changing $y$ should not cause the performance to change much. We can include this requirement as a heuristic by penalizing the variance of $J$. Intuitively, this heuristic "smooths" the cost function with respect to the exogenous parameters: regions of high variance (containing sharp local minima) are penalized, while regions of low variance are rewarded. This heuristic leads us to the *variance-regularized robust design optimization problem*:

$$\min_{x\in\mathcal{X}} \quad \mathop{\mathbb{E}}_{y\sim\mathcal{Y}}\Big[J\circ S\left(x,y\right)\Big] + \lambda \mathrm{Var}_{\mathrm{y}\sim\mathcal{Y}}\Big[\mathrm{J}\circ\mathrm{S}\left(\mathrm{x},\mathrm{y}\right)\Big] \tag{A.1a}$$

Practically, we replace the expectation and variance with unbiased estimates over $N$ samples

$y_i \sim \mathcal{Y}, i = 1, \ldots, N.$

$$\min_{x \in \mathcal{X}} \quad \frac{1}{N} \sum_{i=1}^{N} \left[ J \circ S\left(x, y_i\right) \right] + \lambda \left[ \frac{\sum_{i=1}^{N} \left( J \circ S\left(x, y_i\right) \right)^2}{N-1} - \frac{\left( \sum_{i=1}^{N} J \circ S\left(x, y_i\right) \right)^2}{(N-1)N} \right] \tag{A.2a}$$

**Efficiency** Of course, this Monte-Carlo estimate of the variance will require multiple evaluations of $J \circ S$ (where $\circ$ denotes function composition) to evaluate (A.2a). Since $S$ might itself be expensive to evaluate, approximating the gradients of (A.2a) using finite differences or a stochastic gradient estimate will impose a large computational cost ($2nN$ additional evaluations of $J \circ S$ and $c_i$ at each step for finite differences). Instead, we can turn to automatic differentiation (AD) to directly compute these gradients with respect to $x$, which we can use with any off-the-shelf gradient-based optimization engine. The precise choice of optimization algorithm is driven by the constraints and is not central to our framework. If $\mathcal{X}$ is defined by hyper-rectangle bounds on $x$, then algorithms like L-BFGS-B may be used, but if the constraints are more complex then sequential quadratic programming or interior-point methods may be used.

In this framework, the user need only implement the simulator and cost function for their specific problem using a differentiable programming framework like the JAX library for Python [Bradbury et al., 2018], and this implementation can be used automatically for efficient gradient-based optimization. By implementing a library of additional building blocks in this AD paradigm (e.g. estimation algorithms like the EKF), we can provide an AD-based design optimization tool that strikes a productive balance between flexibility and ease of use.

## A.2 Details on multi-agent manipulation case study

Consider a case study where two mobile robots must collaborate to move a large object. In this setting, two ground robots must collaborate to push a box from its current location to a target pose (as in Fig. 3.2). Given the desired box pose and the current location of each robot, a neural network plans a trajectory for each robot, which the robots then track using a feedback controller ($x$ includes both the neural network parameters and the tracking controller gains, with a total of 454 design parameters). The exogenous parameters include the coefficient of friction for each contact pair, the mass of the box, the desired pose of the box, and the initial pose for each robot (a total of 13 exogenous parameters; we vary the desired box pose and initial robot poses to prevent over-fitting during optimization). The cost function is simply the squared error between the desired box pose (including position and orientation) and its true final pose after a 4 s simulation. We implement the contact dynamics simulator, trajectory planning neural network, and path tracking controller in Python using JAX.

This case study involves interactions between planning and control subsystems, complicated dynamics, and a high-dimensional design space (¿400 design parameters). This example also showcases a different interpretation of the exogenous parameters: instead of representing true sources of randomness, these parameters represent quantities that are simply unknown at design-time. For example, the target position for the box is not random in the same way as sensor noise in the previous example, but since we cannot choose this value at design-time it must be included in $y$. As a result, minimizing the expected cost with respect to variation in $y$ yields a solution that achieves good performance for many different target poses, enabling the user to select one at run-time and be confident that the design will perform well.

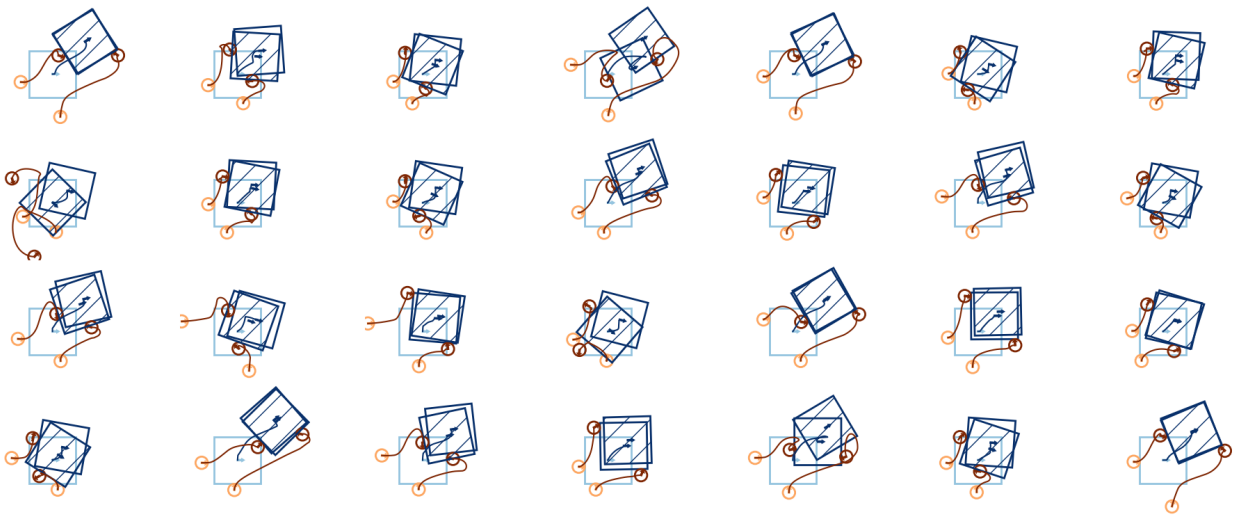


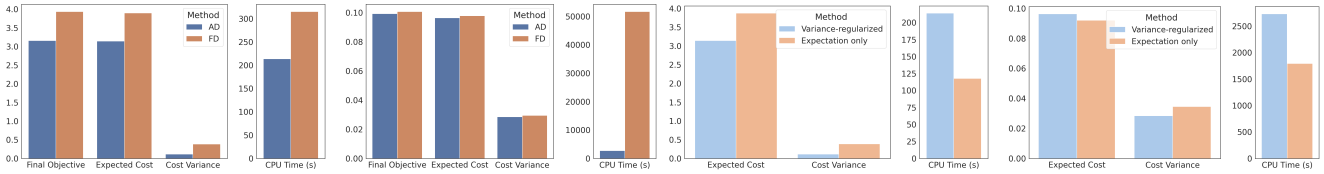
**Figure A.1** | Additional examples of optimized multi-agent manipulation behavior in simulation, showing that the optimized strategy reaches the goal in most cases. Each example shows the results of executing the optimized pushing strategy for 4 s with a randomly selected set of friction coefficients, random target pose, and random initial robot poses. Light/dark colors indicate initial/final positions, and the striped box indicates the target pose.

To solve this design problem, the neural network parameters are initialized i.i.d. according to a Gaussian distribution, and the tracking controller gains are set to nominal values. We then optimize the parameters using $N = 512$, $\lambda = 0.1$, and L-BFGS-B back-end. This optimization took 45 minutes 32 s on a laptop computer with no GPU (8 GB of RAM and a 1.8 GHz 8-core CPU). Fig. 3.2 shows a comparison between the initial and optimized strategies, and Fig. **??** in the appendix shows additional examples of the optimized behavior. The target pose is drawn uniformly $[p_x, p_y, x] \in [0, 0.5]^2 \times [-\pi/4, \pi/4]$, and the optimized design achieves a mean squared error of 0.0964.

We tested the optimized design in hardware, again using the Turtlebot 3 platform. An overhead camera and AprilTag [Olson, 2011] markers were used to obtain the location of the box and each robot. At execution, each robot first moves to a designated starting location near the box, plans a trajectory using the neural network policy, and tracks that trajectory at 100 Hz until the box reaches its desired location or a time limit is reached. Results from this hardware experiment are shown in Fig. 3.2, and a video is included in the supplementary materials. No parameter tuning or estimation was needed to transfer to hardware.

To assess the robustness and efficiency of this design optimization method, we must answer a number of questions. For instance, how does automatic differentiation compare with other methods for estimating the gradient (e.g. finite differences)? What benefit does variance regularization in problem (3.5) bring? We answer these questions here using an ablation study where we attempt to isolate the impact of each of these features.

First, why use automatic differentiation? On the one hand, AD allows us to estimate the gradient with only a single evaluation of the objective function, while other methods (such as finite differences, or FD) require multiple evaluations. On the other hand, AD necessarily incurs some overhead at runtime, making each AD function call more expensive than those used in an FD scheme. Additionally, some arguments [Suh et al., 2021] suggest that exact gradients may be less useful than finite-difference or stochastic approximations when the objective is stiff or discontinuous. We compare AD with a 3-point finite-difference method by re-solving problem (3.5)




**(a)** AD vs. FD; sensor placement **(b)** AD vs. FD; manipulation **(c)** VR; sensor placement **(d)** VR; manipulation

**Figure A.2** | (a)-(b) Improvement of automatic differentiation (AD) over finite differences (FD) in both case studies. (c)-(d) Effect of variance regularization (VR) in both case studies.

for both case studies, keeping all parameters constant ($N = 512$, $\lambda = 0.1$, same random seed) and substituting the gradients obtained using AD for those computed using finite differences. Fig. 3.3 shows the results of this comparison. In the sensor placement example, AD achieves a lower expected cost and cost variance, and it runs in 32% less time. In the collaborative manipulation example, both methods achieve similar expected cost and variance, but the AD version runs nearly 19x faster. These results lead us to conclude that AD enables more effective optimization than finite differences and is an appropriate choice for our framework.

The next question is whether variance regularization brings any benefit to the design optimization problem. To answer this question, we compare the results of re-solving both case studies with variance weight $\lambda = 0.1$ and $\lambda = 0$. These results are shown in Fig. 3.3; surprisingly, in the sensor placement example we see that the variance-regularized problem results in a lower expected cost, contrary to the intuition that regularization requires a trade off with increased expected cost. We expect that this lower expected cost may be a result of the regularization term smoothing the objective with respect to the exogenous parameters. However, these benefits are less pronounced than the benefits from automatic differentiation, and we see more benefit in the sensor-placement example reported in [Dawson and Fan, 2022a] than in the multi-agent manipulation problem.

## A.3 Details on satellite trajectory planning case study

We can express this problem using dynaimcs in the Clohessy-Wiltshire-Hill coordinate frame [Jewison and Erwin, 2016], which assumes that the target's orbit is circular and constructs a coordinate frame with the origin at the target, the $x$-axis pointing away from the Earth, the $y$-axis pointing along the target's orbit, and the $z$-axis pointing out of the orbital plane. In this frame, the chaser's dynamics are approximately linear, with positions $p_x$, $p_y$, $p_z$ and velocities $v_x$, $v_y$, $v_z$ varying according to controlled thrust in each direction $u_x$, $u_y$, $u_z$:

$$\begin{bmatrix} \dot{p}_x \\ \dot{p}_y \\ \dot{p}_z \\ \dot{v}_x \\ \dot{v}_y \\ \dot{v}_z \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 3n^2 p_x + 2n v_y + u_x/m \\ -2n v_x + u_y/m \\ -n^2 p_z + u_z/m \end{bmatrix}$$

$n = \sqrt{\mu/a^3}$ is the mean-motion of the target, determined by the Earth's gravitational constant $\mu = 3.986 \times 10^{14}\,\mathrm{m}^3/\mathrm{s}^2$ and the target's altitude $a$ (i.e. the length of the semi-major orbital axis, 353 km in low Earth orbit). $m = 500\,\mathrm{kg}$ is the mass of the chaser satellite [Jewison and Erwin, 2016].