5-2024

Assessing Extant Methods for Generating *G*-Optimal Designs and a Novel Methodology to Compute the *G*-Score of a Candidate Design

Hyrum John Hansen
Utah State University, a02243798@usu.edu

ASSESSING EXTANT METHODS FOR GENERATING G-OPTIMAL DESIGNS AND A NOVEL METHODOLOGY TO COMPUTE THE G-SCORE OF A CANDIDATE DESIGN

by

Hyrum John Hansen

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTERS OF SCIENCE

in

Statistics

Approved:

Stephen J. Walsh, Ph.D. Major Professor	Brennan Bean, Ph.D. Committee Member
Rong Pan, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice President Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan,Utah

2024

ABSTRACT

Assessing Extant Methods for Generating G-Optimal Designs and a Novel Methodology to Compute the G-Score of a Candidate Design

by

Hyrum John Hansen, Master of Science

Utah State University, 2024

Major Professor: Stephen J. Walsh
Department: Mathematics and Statistics

Finding exact G-Optimal designs has been an outstanding problem in optimal design of experiments since the field's inception. Proposed in 1918, the original G-optimal designs considered only one-factor experimental settings with multi-factor designs precluded due to mathematical intractability. The past two decades have seen rigorous development in the field of numerical optimization, providing the requisite algorithms to find highly G-efficient designed experiments. In this thesis we apply four algorithms to the search for G-optimal designs: the Nelder-Mead Simplex Algorithm, the Particle Swarm Optimizer, and two variants of the Point-Exchange Algorithm. We introduce `Gloptipoly`, a MATLAB implementation of the method of semidefinite relaxations to solve the generalized problem of moments, and use it to score candidate G-optimal designs. We find that combining Particle Swarm Optimization with `Gloptipoly` results in the most G-efficient designs found to-date for several design scenarios. Consistent with past research, our results suggest that swarm optimizers – such as the particle swarm – are the ideal algorithms to search the space of candidate designs. This is because they are robust to entrapment in local optima and are capable of considering many regions of the search space simultaneously. By contrast, greedy algorithms like the very popular coordinate exchange fall short when the dimension of the search space is large. We show that the method of semidefinite relaxations mitigates error induced by approximation methods when computing the G-efficiency of a candidate design.

We also introduce G-optimal designs for response-surface models not-yet explored in the literature, noting interesting geometric features related to how design-points are distributed with respect to their corresponding scaled-prediction-variance surfaces. Computational complexity precludes extension to settings with more than three factors, making an implementation of the method of semidefinite relaxations in a faster language essential to progress in this field. Nonetheless, the proposed designs for cubic, quartic, and higher-order interaction response-surface models are the first known G-optimal designs of their kind.

(80 Pages)

PUBLIC ABSTRACT

Assessing Extant Methods for Generating G-Optimal Designs and a Novel Methodology to Compute the G-Score of a Candidate Design

Hyrum John Hansen

Experimental designs are used by scientists to allocate treatments such that statistical inference is appropriate. Most traditional experimental designs have mathematical properties that make them desirable under certain conditions. Optimal experimental designs are those where the researcher can exercise total control over the treatment levels to maximize a chosen mathematical property. As is common in literature, the experimental design is represented as a matrix where each column represents a variable, and each row represents a trial. We define a function that takes as input the design matrix and outputs its score. We then algorithmically adjust each entry until a design is found that minimizes or maximizes the function of interest.

In this thesis we study how to best minimize a prediction-variance criteria called the G-criteria, which is the maximum scaled-prediction-variance (SPV) over the entire design-space. Researchers may choose to implement this type of optimal experimental design when they need make accurate predictions on untested regions of their design space. In this thesis we apply various algorithms to the G-optimal problem, identifying and scoring candidate G-optimal designs with a combination of novel and legacy algorithms. We find that combining a novel scoring method called `Gloptipoly` with the best-known searching method (particle swarm optimization) produces the best-known G-optimal designs to-date.

CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
REVIEW OF G OPTIMAL DESIGNS ON THE HYPERCUBE

1.1
Introduction

Optimal experimental designs are found by minimizing or maximizing a specific function of the model information matrix. Optimal designs can offer distinct advantages, especially when experimental constraints necessitate efficiency and when the researcher can predefine the response surface model. For example, situations characterized by budgetary constraints, which limit the number of experimental trials, are especially well-suited for optimal designs. In such cases, researchers might focus on tailoring their experiments to optimize objective measures of quality within the confines of limited trials, rather than attempting to conform the experiment to classical designs.

The concept of optimal designs was originally introduced by Kristine Smith in 1918. Her seminal paper provided guidelines for constructing experiments by utilizing residual error variance where the collected data are used to fit a polynomial model with linear parameters. In her study, she examined one-factor designs ranging from first-order to sixth-order polynomials. Smith's paper presented designs that minimize the worst-case-scenario prediction variance, a property we now refer to as the G-criterion. Computational intractability limited Smith to univariate experiments as she was required to analytically derive them using scalar notation. Even for the simplest cases, this was hardly a trivial endeavor. Notwithstanding its limited scope, this paper laid the groundwork for the study of optimal experimental designs and is considered to be 30 years before its time [28].

In the 1950s, the field saw significant theoretical development to address the nonoptimality of classical designs with significant contributions from Kiefer and Wolfowitz [11, 12, 13]. The field's focus was on continuous optimal designs—also called approximate or asymptotic designs—which consider properties of the design achieved for a very large number of trials. These designs are viewed as probability measures on the space of all possible designs and provide convenient mathematical properties as well as some theorems for validating a design's optimality.

In contrast, exact optimal designs are tailored to be optimal for a specific number of trials. Exact optimal design theory lacks analogous mathematical tractability and corresponding theorems for validating a candidate design, so the implemented design is compared to a large group of candidate optimal designs. Research in this space has focused on efficient ways to explore the space of candidate designs, which is exceedingly vast for experiments involving numerous variables. The most popular algorithm for searching this space is the coordinate-exchange (CEXCH) algorithm of Meyer and Nachtsheim [19], an iterative optimization algorithm that updates one coordinate of a design at a time while holding others fixed to improve design quality at each update. For additional examples of research in this space, see [29, 31, 30, 21, 3, 5]

While CEXCH has been relatively successful, it is not without drawbacks. CEXCH begins with a single random candidate design and iteratively moves towards a local optimum, rendering it highly sensitive to the initial condition. Goos & Jones (2011) suggest mitigating the influence of poor starting points by running the algorithm several thousand times with different starting points and choosing the best design from the candidate pool [4]. Because CEXCH is a local optimizer it is unlikely—especially in higher dimensional problems—that it will find the globally optimal design. To overcome these limitations, researchers have implemented meta-heuristic evolutionary algorithms for optimal design generation that have desirable properties for high-dimensional problems like these. These algorithms have included the Genetic Algorithm (GA), Simulated Annealing, and Particle Swarm Optimization (PSO). Borkowski gave the first treatment of the genetic algorithm applied to the optimal design problem in [1, 32, 23].

There are three primary reasons researchers have chosen to explore this class of algorithms for the optimal design problem. First, few—if any—assumptions about the nature of the objective make them agnostic to its peculiarities. These peculiarities will be discussed at length in this and the following chapter, but for now it is sufficient to highlight the fact that since the G-criterion is an optimization problem, the search for G-optimal designs is viewed as a saddle-point optimization with a non-differentiable objective. Second, they overcome CEXCH's propensity to entrapment in local optima through techniques like swarm intelligence. For example, instead of starting with one randomly generated candidate design as in CEXCH, PSO begins with dozens or hundreds of candidate designs and uses information from current and prior positions of all candidate optima in the update equation. Third, this class of algorithms propose updates without needing to compute the gradient of the objective function making it well suited to the optimal design problem.

1.2

Notation

Since Smith's 1918 paper [25], the notation and terminology associated with the field have been rigorously updated. In 1959, Kiefer formalized the field of optimal design, formalizing several scalar measures related to desirable properties of designed experiments [11]. We will not provide an exhaustive list of these measures, but we will borrow much of the notation and terminology. We will be introducing our notation here through the framework of the second-order model. Let N represent the number of design points and K represent the number of experimental factors. Use $\mathbf{x}^{'}$ to denote a design point, a $1 \times K$ row vector where each $x \in \mathbf{x}^{'}$ specifies a factor level. Standard practice is to scale the feasible domain to $[-1, 1]$ so the space of all $\mathbf{x}^{'}$ design points is the $\chi = [-1, 1]^K$ hypercube. This notation is used to denote the Cartesian product of K such elements, which means that resultant mathematical object is a row vector of K elements where each element $x_j \leq |1|$. This abstract object represents a design point, which is realized as a row in the design matrix. The collection of N design points as rows in a matrix engenders the design matrix $\mathbf{X}$, an NK-dimensional hypercube. We use $\times_{j=1}^{N}$ to represent the Cartesian product of N items, which can also be used to abstractly represent the design matrix $\mathbf{X}$. Putting it all together,

$$\mathbf{X} \in \times_{j=1}^{N} \chi = \times_{j=1}^{N} [-1, 1]^K = [-1, 1]^{NK} = \chi^N.$$

The second-order linear model contains $\binom{K+2}{2}$ linear coefficient parameters and is written in scalar form

$$y = \beta_0 + \sum_{i=1}^{K} \beta_i x_i + \sum_{i=1}^{K-1} \sum_{j=i+1}^{K} \beta_{ij} x_i x_j + \sum_{i=1}^{K} \beta_{ii} x_{ii}^2 + \epsilon,$$

or as

$$\mathbf{y} = \mathbf{F}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

in matrix-vector form. The standard ordinary least squares assumptions are imposed; $\boldsymbol{\epsilon} \sim \mathcal{N}_N(\mathbf{0}, \sigma^2 \mathbf{I}_N)$ where $\mathcal{N}_N$ is the N-dimensional multivariate normal distribution. $\mathbf{F}$, the $N \times p$ model matrix, is obtained from $\mathbf{X}$ through $\mathbf{F}(\mathbf{X})$, and rows in $\mathbf{F}$ are given by the expansion vector

$$\mathbf{f}'(\mathbf{x_i'}) = (1 \quad x_{i1} \quad \ldots \quad x_{ik} \quad x_{i1}x_{i2} \quad \ldots \quad x_{i(K-1)}x_{iK} \quad x_{i1}^2 \quad x_{iK}^2).$$

$\mathbf{F}(\mathbf{X})$ and $\mathbf{F}$ are used interchangeably in literature, but it should be clear that $\mathbf{F}$ is always a function

of $\mathbf{X}$.

The ordinary least squares estimator of $\boldsymbol{\beta}$ is $\hat{\boldsymbol{\beta}} = (\mathbf{F}'\mathbf{F})^{-1}\mathbf{F}'\mathbf{y}$, which has variance $\text{Var}(\hat{\boldsymbol{\beta}}) = \sigma^2(\mathbf{F}'\mathbf{F})^{-1}$, also called the full information matrix. Both $\hat{\boldsymbol{\beta}}$ and $\text{Var}(\hat{\boldsymbol{\beta}})$ are expressed as quantities of $\mathbf{F}'\mathbf{F}$, the model information matrix. Because of its centrality to the optimal design problem, let $\mathbf{M}(\mathbf{X}) = \mathbf{F}'\mathbf{F}$. Most optimal design objective criteria are dependent on total information matrix (and all traditional criteria are invariants of the information matrix), so it is of considerable importance to the field [4].

We use $\mathbf{M}(\mathbf{X})$ to formulate objective criteria, functions of the information matrix whose output describes some property of the candidate design that would be expedient to minimize or maximize. Typically, the practitioner must choose exactly one $\mathbf{X} \in \chi^N$, and objective criteria offer meaningful ways of measuring the quality of a candidate design from a large candidate pool. In order to find an optimal design, we need three pieces of information. First, the number of experimental trials, N, must be given. Second, the response surface model with which the practitioner intends to analyze the data must be known. Finally, we must define the optimality criteria, which encodes the desirable experimental design property.

1.3

G-Optimality

There are two prediction variance criteria we commonly study. These are referred to as the I- and G-criteria, where the I-optimal design minimizes the average prediction variance and the G-optimal design minimizes the worst-case-scenario prediction variance. This thesis focuses on G-optimal designs. G-optimal designs have both intuitive and practical appeal, and are a good choice when prediction on untested regions is a primary objective of experimentation. A practitioner should favor the G-optimal design over the I-optimal design when prediction accuracy on untested regions is paramount. It should also be noted that hybrid designs exist; that is, some work has been done to find designs that are simultaneously both highly G- and I-efficient [26]. Nonetheless, this work focuses exclusively on optimizing for G-optimality and will not consider other design types. Intuitively, the G-optimal is that design which minimizes the worst-case-scenario prediction variance over χ. We compute the variance of the mean predicted value with

$$\text{Var}(\hat{y}(\mathbf{x}')) = \sigma^2\mathbf{f}'(\mathbf{x}')(\mathbf{F}'\mathbf{F})^{-\mathbf{1}}\mathbf{f}(\mathbf{x}).$$

Because we need this function to provide a numeric value to guide the optimization and σ^2 is

calculated using actual numeric data, we multiply by the variance by a scaling factor, N/σ^2, and use Scaled Prediction Variance (SPV). With this new expression, we can compute scaled prediction variance for any design point $\mathbf{x}' \in \chi$. The resulting quantity is expressed as

$$\text{SPV}(\mathbf{x}'|\mathbf{X}) := N\mathbf{f}'(\mathbf{x}')(\mathbf{F}'\mathbf{F})^{-\mathbf{1}}\mathbf{f}(\mathbf{x}).$$

The G-score of a candidate design is the maximum scaled prediction variance over all design points, that is

$$\mathbf{G}(\mathbf{X}) := \max_{\mathbf{x}' \in \chi} \text{SPV}(\mathbf{x}'|\mathbf{X}).$$

As stated, computing the G-score is itself an optimization problem making the search for a G-optimal matrices a nested optimization problem. So the G-optimal design $\mathbf{X}^*$ is that design which minimizes, over all $\mathbf{X} \in \chi^N$, the maximum prediction variance. We can formulate the search as a minimax problem, that is

$$\begin{aligned}\mathbf{X}^* &:= \operatorname*{argmin}_{\mathbf{X} \in \chi^N} G(\mathbf{X}) \\ &= \operatorname*{argmin}_{\mathbf{X} \in \chi^N} \max_{\mathbf{x}' \in \chi} \text{SPV}(\mathbf{x}'|\mathbf{X}).\end{aligned}$$

Neither of these optimizations is convex, making the search for G-optimal designs a notoriously difficult saddle-point problem. Fortunately, there exists a theoretical lower bound on the G-score of candidate matrices, so we have a baseline for design quality before the search even begins. Outlined in [11], the general equivalence theorem establishes p, the parameter count, as the minimum threshold. In other words,

$$G(\mathbf{X}) = \max_{\mathbf{x}' \in \chi} \text{SPV}(\mathbf{x}'|\mathbf{X}) \geq p.$$

Note that not all design scenarios admit designs that actually achieve this lower bound, but it does provide a convenient way to check if generated designs are close to being G-optimal. Exploiting the fact that the theoretically globally G-optimal design has $\text{SPV}(\mathbf{x}'|\mathbf{X}) = p$ at all diagonals of the hat matrix $\mathbf{F}(\mathbf{F}'\mathbf{F})^{-\mathbf{1}}\mathbf{F}'$ and $\text{SPV}(\mathbf{x}'|\mathbf{X}) \leq p$ at all other design points $\mathbf{x}' \in \chi$, we can compute an *absolute* G-efficiency score,

$$G_{eff}(\mathbf{X}) = 100\frac{p}{G(\mathbf{X})}.$$

The existence of a known theoretical bound makes the G-optimal problem unique from other exact optimal design problems where the scale on the optimality criterion is arbitrary. To compare the relative quality of two designs, standard practice in optimal design is to report *relative* efficiency, which involves the ratio of the found design to the best-known design, and can be computed as

$$G_{releff}(\mathbf{X_1}, \mathbf{X_2}) = 100\frac{G_{eff}(\mathbf{X_1})}{G_{eff}(\mathbf{X_2})}.$$

1.4

On Generating Exact G-Optimal Designs

When Smith first introduced the optimal design problem in 1918, problems were necessarily solved by hand. This restricted researchers to low-order, univariate models. The age of computing has lowered the barrier to entry, particularly for computationally expensive tasks like computing the G-criteria. Borkowski in [1] was the first to apply the genetic algorithm (GA) to the optimal design problem. The algorithm was adapted to find $A-$, $D-$, $I-$, and $G-$ optimal designs for design scenarios with up to three variables over a range of experiment sizes. Borkowski's designs have been used as a baseline of comparison when evaluating the efficacy of proposed algorithms [19, 1, 23].

In 2010, Rodriguez et al. implemented Brent's minimization algorithm to propose coordinate swaps inside CEXCH and to compute the G-score of a candidate design [22], finding designs on par with those in Borkowski, 2003 [1]. [22] even found the best-known designs for the $K = 4$, $N = 15$ and $K = 5$, $N = 21$ cases. Importantly, they also introduced the Variance-Ratio Fraction of Design Space Plot (VRFDS), a graphical method to compare the prediction variance of competing designs over regions of interest. The authors found that I-optimal designs have significantly lower prediction variance than corresponding G-optimal designs over most of the design space, with the intuitive exception being in the worst case scenario. The I-criteria is to minimize the *average* prediction variance, so its elevated practicability came as no surprise. Nonetheless, the G-criteria could be desirable for models known to be particularly unstable at the extremes and it remains a challenging but academically interesting problem, so the research has persisted.

Saleh & Pan recognized the need for further algorithm development in the G-opt space. In 2015 they published a modified version of the coordinate exchange that integrates a clustering

step [23]. The algorithm determines cluster membership for each design point in the current design matrix. The cluster with the smallest maximum prediction variance is denoted MU while the cluster with the greatest maximum prediction variance is denoted MX. Within MU, points are iteratively removed and resulting maximum prediction variance calculated. The element that causes minimal change in maximum prediction variance is then deleted and replaced by the candidate point with maximum prediction variance in the MX cluster. The algorithm continues to cluster, evaluate, and replace until no changes are made. Note that this procedure can't work for a fully saturated model because it relies on computing the maximum prediction variance for an $(N-1) \times p$ matrix, which is singular; however, for viable experimental settings clustering-based coordinate exchange was shown to produce design with higher G-efficiency in much shorter time when compared to previous methods. In 2017, Hernandez et al were able to exploit a clever transformation, converting the unwieldy G-optimal search to a more computationally tractable I_λ search [8]. The authors used the standard coordinate exchange algorithm with a continuous I_λ-optimal design as a starting point. They refer to method as $G(I_\lambda)$-CEXCH. The method proved to be fairly effective and computationally inexpensive, at least compared to GA. The designs found by $G(I_\lambda)$-CEXCH were greater than 90% efficient and about 2 orders of magnitude less computationally expensive than GA. This result made it a great choice to find G-optimal designs. A tabular summary of published work on the G-optimal problem can be found in Table 1.1.

Walsh & Borkowski were the first to apply particle swarm optimization (PSO) to the G-optimal problem [32]. A version of PSO that utilizes a local communication topology was shown to be an effective algorithm to find designs under the $D-$ and $I-$ criteria, but its usefulness hadn't yet been evaluated for the G-optimal problem. Despite being a relative newcomer, PSO produced the best-known designs to-date with cost commensurate to $G(I_\lambda)$-CEXCH making it the best known choice for finding G-optimal designs. PSO allows information about the location of the current best-known design to influence the behavior of all other designs in the search, which gives it a significant advantage over CEXCH whose runs are completely blind to the information gained by other trials. PSO also doesn't require repeated runs like CEXCH, as 200-runs of the algorithm is analogous to a 200-particle PSO search.

Table 1.1: Summary of design scenarios, algorithms, and authors who have published resutls for the G-optimal design problem for a second order response surface model in the last 20 years [1, 8, 32, 22, 23].

# Exp. Factors K	**Experiment Sizes** N	**Algorithms**	**Authors**
1	3, 4, 5, 6, 7, 8, 9	GA	Borkowski (2003)
		$G(I_\lambda)$-CEXCH	Hernandez and Nachtsheim (2018)
		G-PSO	Walsh and Borkowski (2022)
2	6, 7, 8, 9, 10,11, 12	GA	Borkowski (2003)
		CEXCH	Rodriquez et. al. (2010)
		cCEA ($N = 7$ to 12)	Saleh and Pan (2015)
		G-PSO	Walsh and Borkowski (2022)
3	10, 11,12,13,14,15,16	GA	Borkowski (2003)
		CEXCH	Rodriquez et. al. (2010)
		cCEA ($N = 11$ to 16)	Saleh and Pan (2015)
		G-PSO	Walsh and Borkowski (2022)
4	15, 20, 24	CEXCH	Rodriguez et. al. (2010)
		cCEA (N=24)	Saleh and Pan (2015)
	16	cCEA	Saleh and Pan (2015)
	17	$G(I_\lambda)$-CEXCH	Hernandez and Nachtsheim (2018)
	15, 17, 20, 24	G-PSO	Walsh and Borkowski (2022)
5	21, 26, 30	CEXCH	Rodriquez et. al. (2010)
		cCEA ($N = 26$)	Saleh and Pan (2015)
	23	$G(I_\lambda)$-CEXCH	Hernandez and Nachtsheim (2018)
	21, 23, 26, 30	G-PSO	Walsh and Borkowski (2022)

1.5

Computing the G-Score

Computing the G-score for a candidate design is by no means a trivial endeavor. In Chapter 3 the process by which we find and propose optimal designs will be described in more detail, but for readability we will take a moment here to define some key terms. In this work, when we refer to the 'outer-optimization,' we are talking about the multivariate optimization step that takes as input an $N \times K$ matrix and adjusts each entry until convergence. In the context of element-by-element adjustment, convergence means a practitioner can pass their univariate optimizer across all $N \times K$ elements in the matrix and no updates will be proposed. There are also multivariate optimization algorithms for which convergence usually means the outer-optimizer has reached some specified level of tolerance regarding the difference between a proposed optimum at time step t vs. $t-1$, but these processes will be discussed as needed later in the work. Accordingly when we refer to the 'inner-optimization,' we are talking about computing the G-efficiency of a given design which is the maximum value on that design's SPV polynomial. So the outer optimizer searches the space of candidate designs and proposes updates using the inner optimization as its objective function. In this work, we are primarily studying the inner-optimization step. Similarly, the SPV

polynomial, referred to frequently in this work as the SPV surface, is the K-dimensional scalded-prediction-variance polynomial engendered by a candidate design. This polynomial outputs the scaled prediction variance for any design point, so its maximum value is the quantity we aim to minimize when computing the G-criterion. These ideas will be discussed at length later, but a brief explanation is warranted here before we get into detail.

Various methods have been employed to approximate the G-score of a candidate design matrix, but there is plenty of space to explore new techniques. The 'inner-optimization' can be one of the trickiest steps in the search. A common practice is to exploit the symmetry of the SPV-surface for the second-order RSM model and use a 5^K grid-approach over χ. Each $x_i \in \{-1, -0.5, 0, 0.5, 1\}$ for $i = 1, ..., K$ with the full grid being $G_\chi = \{-1, -0.5, 0, 0.5, 1\}^K$ meaning that the approach requires 5^K function evaluations *at each iteration*. For vanilla CEXCH, this method demands $5^5 = 3,125$ function evaluations per proposed swap for a $K = 5$ experimental setting. For a $K = 5, N = 30$ experimental setting, one pass of a univariate optimizer through the design matrix requires $N \times K \times 3,125 = 30 \times 5 \times 125 = 468,750$ function evaluations *per pass*, *per design*. For high-dimensional problems optimizers usually take more than 1 pass through the matrix to converge to a local optimum, so scoring a candidate design with the grid-search can require over a million function evaluations just to propose a single candidate design. Nonetheless, both [10, 11] used this procedure with remarkable success, due largely to the symmetry of the quadratic G-surface reporting small errors and finding highly G-optimal designs. It should be noted that a grid approximation is only reliable for models whose response-surface model engenders symmetric SPV surfaces (more on this in Chapter 4). Researchers have utilized other methods to score candidate designs including Brent's minimizer in [23], but using algorithms that don't guarantee global optima can result in a seriously overstated claim about the G-optimality of a given design. Put differently, an underestimate of max SPV is akin to an overestimate of design quality. Since it is important that a practitioner understand fully the theoretical implications of their design, an accurate scoring mechanism is imperative. Hence, there remains need for a robust, reliable way to compute the G-score of a candidate design.

The novelty of this project surrounds the utilization of a technique known to find the global maximum of a polynomial. The technique is known as the method of semidefinite relaxations [15] and there exists a convenient MATLAB implementation. As will be clearly articulated in Chapter 2, the SPV surface is a $2 * r$-order polynomial where r is the order of the specified RSM model. Many conventional optimization routines are known to be prone to entrapment in local optima. Since every design-scenario engenders a unique SPV-surface, computing the true maximum for every design covered during the search can be difficult. To better understand the problem, consider

Figure 1.1 where three different SPV-surfaces are visualized alongside the true maximum value on each polynomial. If an off-the-shelf optimizer is applied to the problem of computing the G-score of a candidate design, it may get stuck in a local optima and fail to capture the true maximum.

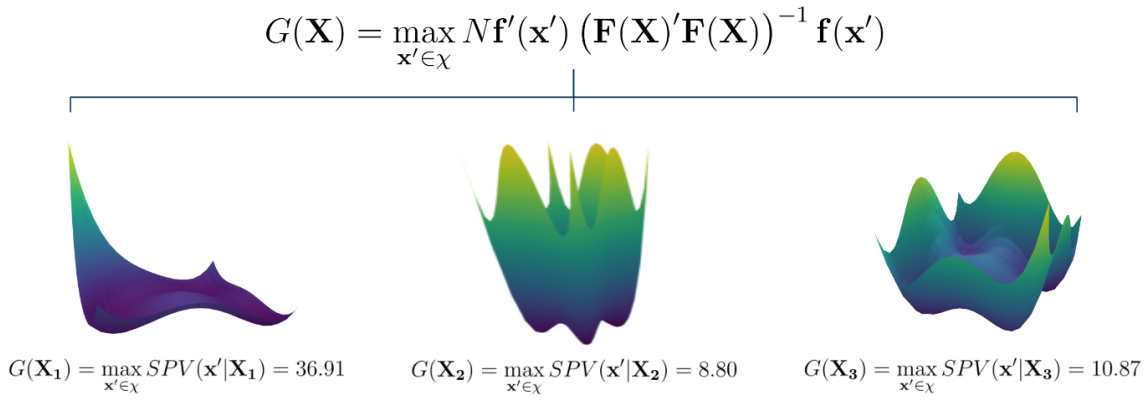

Figure 1.1: Three unique SPV-surfaces engendered by the second-order RSM model for a two-factor, 8-trial experimental setting. Max-SPV is enumerated below each surface visualization.

As we discuss in Chapter 4, there are also cases where the locations sampled by a grid approximation don't adequately sample the SPV-surface, resulting in a misscored design.

1.6 Conclusion

Walsh & Borkowski produced the best-known designs by combining PSO with the aforementioned 5^K grid-search approach to scoring a candidate design [32]. Naturally, a grid search approach induces some error, though the symmetry of the SPV surface engendered by the full, second-order model helps mitigate the method's imperfections. Nonetheless, finding an algorithm that can compute an exact G-score would represent significant progress in the field. As will be shown in the next section, the function we optimize to compute the G-criteria is a fourth-order polynomial, so a polynomial optimizer is sufficient to find the correct scores. Finding the global optimum of a polynomial has been explored in industrial engineering literature, but has not yet been applied to the G-optimal problem. The following chapters explore a promising algorithm [7] to help solve the G-optimal design generation problem.

The remainder of this thesis proceeds as follows. First, we will introduce the generalized problem of moments and explore using the method of semidefinite relaxations to find the global optimum of a polynomial, which is the geometric form of an SPV surface. We will then explore

`Gloptipoly`, a MATLAB routine used to solve create semidefinite relaxations and `SeDuMi`, the package we use to find their optima [6, 7]. Following the software introduction, we pair `Gloptipoly` with the Nelder-Mead Simplex Algorithm to find highly G-efficient designs for all the cases covered in [1]. We then extend our search to higher-order models not yet explored in literature. Next, point-exchanges are implemented and tested to stay consistent with common methods in thie field. Finally, we pair PSO with `Gloptipoly` and find G-optimal designs with higher G-efficiency than any others propose in the literature for several design cases. We conclude by comparing all tested methods in this work, and propose some future research directions.

CHAPTER 2
PROBLEM OF MOMENTS, GLOBAL OPTIMIZATION WITH POLYNOMIALS, AND GLOPTIPOLY

2.1
The Generalized Problem of Moments

`Gloptipoly` was developed for solving, and in more complex situations, approximating the Generalized Problem of Moments (GPM). The GPM has its origins in the early works of renowned mathematicians such as Chebyshev, Markov, and Stieltjes. [17]. It is considered to be an infinite-dimensional optimization problem and has been rigorously studied in various fields including probability, finance, optimization, signal processing, chemistry, and tomography. For a complete discussion of methodologies and applications, see [14]. A reliable way to solve the GPM is of interest to the field of optimal design because computing the G-score is a special case of the GPM, as will be shown in subsequent sections.

Algorithmic development for solving the Generalized Problem of Moments (GPM) has been a central area of focus in optimization research since the late 20th century. Hernandez-Lerma & Lasserre introduced a method for using approximation schemes to address infinite linear programs. Their work demonstrates that under certain assumptions, optimal solutions to a GPM can be approximated by finding the optimal solution to each of a sequence of finite-dimensional linear programs associated with the original problem. These are sometimes called linear program (LP) relaxations [9]. Their technique takes an infinite-dimensional linear program and discritizes it, breaking the problem up into finite-dimensional linear programs of increasing size. They demonstrate that an accumulation point for a sequence of optimal solutions to one of the finite-dimensional approximating programs is also an optimal solution for the original infinite-dimensional program. This is quite a powerful result, and it informed the theory presented in [15] which expanded the idea of finding global polynomial optima by reducing the problem of moments to a sequence of convex linear matrix inequality (LMI) problems. There are, however, some issues with using LP relaxations to solve

the GPM. Numerical instability and the absence of a general convergence guarantee make the LP relaxation technique inappropriate for finding optima of less well-behaved functions. To overcome these challenges, semidefinite program (SDP) relaxations were introduced [16]. SDP relaxations offer an attractive solution for small-scale problems because they include a convergence guarantee and resolve some numerical stability issues, but they have not yet been implemented to solve very large optimization problems. They also impose an additional constraint on the objective function: it must be a semidefinite polynomial. So for the practitioner, both LP and SDP are useful tools, but SDP generally outperforms LP for small scale and medium problems making it the technique of choice to find max SPV.

2.2

The Development of `Gloptipoly`

Using the theory developed by [16], an algorithm was proposed to formulate and solve convex linear matrix inequality relaxations of the global optimization problem of minimizing a polynomial function with polynomial inequality, equality, or integer constraints. This algorithm, called `Gloptipoly`, generates a series of lower bounds that monotonically converge to the global optimum [6]. The original version, implemented in MATLAB, was restricted to problems with few dimensions and could only build a hierarchy of LMI relaxations of the GPM. Theory from [17] was harnessed in the implementation of `Gloptipoly3`, expanding the software's capability to use SDP relaxations to solve the GPM [7]. It should be noted that `Gloptipoly` is not a standalone GPM solver; rather, it converts the original problem into a sequence of SDPs that a user must solve by calling an external semidefinite solver. `SeDuMi` [27] is the `Gloptipoly` default and seems to be widely used, but the software is configured to work with any other semidefinite solver in YALIMP, a MATLAB toolbox for interfacing SDP solvers [2].

The algorithm begins by characterizing the general convex, quadratic, quadratically constrained problem for both the primal and dual problems. Exact LMI relaxations are then formulated for both the primal and dual problems. The optimal value is obtained at a particular LMI relaxation, and the solution is then output for a user. Accordingly, formulating and solving so many problems makes `Gloptipoly` a computationally expensive algorithm. First, `Gloptipoly` must formulate multiple relaxations for the SPV polynomial. `Gloptipoly` also formulates both dual and primal optimization problems for each semidefinite program to certify optimality. Then `SeDuMi` must solve each of those problems and find the accumulation point for the set of optimal solutions.

This procedure is repeated each time a new candidate design is considered, which corresponds to a single update step of the outer-optimzer. Hence, utilizing efficient outer-optimization schemes is of utmost importance.

2.3

`Gloptipoly` and the SPV Polynomial

As discussed in Section 1.4, several methods have been applied to compute the G-score of a candidate design matrix including Brent's optimizer and a grid search approach. To our knowledge, an algorithm like `Gloptipoly` has not yet been applied to research in this space. `Gloptipoly` is a good candidate for this problem because it is known to be a global polynomial optimizer, and the G-score is nothing more than the global optimum of a polynomial. For illustration purposes, consider the general case in which $(\mathbf{F}'\mathbf{F})^{-1} \in \mathbb{R}^{3\times 3}$. This information matrix corresponds to any experimental setting with 3 trials, which is only advisable for the univariate case due to model saturation for all other response-surface settings. So the design matrix would look something like

$$\mathbf{X} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

where we have one factor x with levels a, b, c. After expanding this vector into the model matrix we're left with a $\mathbf{F} \in \mathbb{R}^{3\times 3}$

$$\mathbf{F} = \begin{bmatrix} 1 & a & a^2 \\ 1 & b & b^2 \\ 1 & c & c^2 \end{bmatrix}.$$

Calculating $\mathbf{F}'\mathbf{F}$ is straightforward, and a formula exists for finding the general inverse of a 3×3 matrix. However, since these computations are not pertinent to this work, they will not be presented here. Without loss of generality, let's simply consider

$$(\mathbf{F}'\mathbf{F})^{-1} = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}.$$

Then we may use the inverse information matrix to compute the SPV.

$$SPV(\mathbf{x}'|\mathbf{X}) = 3 \times \begin{bmatrix} 1 & x & x^2 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}$$

$$= 3 \times \begin{bmatrix} f_{11} + f_{21}x + f_{31}x^2 & f_{12} + f_{22}x + f_{32}x^2 & f_{13} + f_{23}x + f_{33}x^2 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}$$

$$= f_{11} + f_{21}x + f_{31}x^2 + x(f_{12} + f_{22}x + f_{32}x^2) + x^2(f_{13} + f_{23}x + f_{33}x^2)$$

$$= f_{11} + x(f_{21} + f_{12}) + x^2(f_{31} + f_{22} + f_{13}) + x^3(f_{32} + f_{23}) + x^4(f_{33})$$

which is a fourth-order polynomial, making $SPV(\mathbf{x}'|\mathbf{X})$ the type of function `Gloptipoly` is designed to formulate semidefinite relaxations for. As outlined in Chapter 1, the G-score is considered to be the global maximum value on the SPV polynomial. Thus, there is adequate theoretical motivation to apply `Gloptipoly` for scoring a candidate design.

2.4

`Gloptipoly` and the G-Score

Though `Gloptipoly` has desirable theoretical properties, there can be challenges when taking a problem from an abstract mathematical space to its numerical implementation. In order to justify its use as the objective function in a search for highly G-efficient designs, it must first be validated as a tool that can accurately compute the G-criteria of a candidate design. To this effect, it must be tested on designs with known scores. In this validation study we use designs found by [32], whose G-criteria were computed via grid approximation. Their scoring method exploited the symmetry of the SPV function and returned the maximum SPV for $\mathbf{x}' \in \{-1, -0.5, 0, 0.5, 1\}^K$. Table 2.1 gives side by side comparisons of the absolute efficiencies computed using a grid approximation and computed with `Gloptipoly`. There are discrepancies between scores obtained by the two methods, and these discrepancies suggest need for an improved scoring mechanism. A dense grid approximation was also computed to confirm which method is more accurate. We scored designs with the grid $\mathbf{x}' \in \{-1, -0.99, -0.98, ..., 0.98, 0.99, 1\}^K$ and the scores reported by `Gloptipoly` were found to be in agreement with those reported by the dense grid (± 0.01). The denser the grid, the more accurate the reported max SPV, so the agreement provides further evidence that `Gloptipoly` is a good tool

to compute G-efficiencies.

K	N	Grid Approximation	`Gloptipoly`	Absolute Difference
1	3	100	100	0
1	4	82.92	82.92	0
1	5	80.58	80.58	0
1	6	100	100	0
1	7	91.17	91.17	0
1	8	89.13	89.13	0
1	9	100	100	0
2	6	75.03	74.39	0.64
2	7	80.24	80.04	0.19
2	8	87.94	87.94	0
2	9	86.63	84.03	2.60
2	10	87.40	86.30	1.10
2	11	87.07	86.66	0.41
2	12	88.17	88.11	0.06
3	10	71.43	70.38	1.05
3	11	80.51	79.54	0.97
3	12	83.35	83.12	0.22
3	13	86.46	85.81	0.64
3	14	89.71	89.09	0.61
3	15	85.99	85.77	0.22
3	16	85.79	85.39	0.39
4	15	71.09	70.64	0.44
4	17	73.90	73.66	0.24
4	20	80.20	79.31	0.89
4	24	85.95	85.85	0.10
5	21	68.67	67.84	0.83
5	23	73.19	72.67	0.52
5	26	75.31	74.84	0.47
5	30	76.16	75.71	0.45

Table 2.1: Relative efficiencies as reported by the grid approximation and Gloptipoly for all second-order design scenarios covered by [32]. The grid approximation tends to overestimate the G-efficiency for a candidate design when compared to `Gloptipoly`, but the difference is generally marginal. For some cases, the $K = 2$, $N = 9$ design scenario being a notable example, the difference sufficiently pronounced to consider switching algorithms.

To better understand the need for a consistent, accurate scoring mechanism, plots were produced for various SPV surfaces. Based on our studies, the G-optimal design search seems to flatten a candidate design's SPV surface (i.e. the difference between the functions minimum and maximum value tends to decrease as a design approaches a local optimum). This is illustrated by the wide range of SPV values for randomly generated designs and comparatively small range for optimal designs. We expect the absolute difference between the G-efficiency of a candidate optimal design scored with a grid approximation to be larger for random designs and smaller for optimal designs. This will be explored further in Chapter 4 for several response surface models, but here we visualize the SPV surfaces for several $K = 2$ design scenarios using the full second order model to provide

some intuition about the problem.

Figure 2.1 provides SPV surfaces for two experimental designs, one where the input matrix is the G-optimal design for $K = 2, N = 9$ proposed by [32] (left), and the other where each factor level is drawn randomly from $[-1, 1]$ (right). The optimal design engenders an SPV-surface with a fairly small difference between minimum and maximum value compared to the random design. SPV for the completely random design achieves a minimum value of 2.65 and a maximum value of 2602.96 while SPV for the optimal design has a minimum of 3.31 and a maximum of 7.14. So in optimizing a design to minimize the maximum prediction variance across the design space, we have increased the minimum prediction variance while dramatically decreasing the range of values. We would then say that the random design has a max-SPV – or G-score – of 2602.96 with a corresponding G-efficiency of about .27%. The optimal design is used as a baseline for comparison and as such has a G-efficiency of 100%. As we would expect for a completely random design, this is nowhere close to G-optimal. To accommodate the large range of SPV values, SPV for the random design is presented on a natural logarithm scale so that a viewer may note real variation across the surface. Grid points are given in yellow and denote the locations on the SPV surface that would be sampled by a researcher using a grid approximation, returning the maximum value of the 25 tested locations. For both plots, the location of the true optimum (computed with `Gloptipoly` and validated with a dense grid-approximation) is identified by a large red X.

Importantly, the optimum of the G-optimal design is not particularly close to a grid point, further justifying the use of `Gloptipoly` to aid in the search for G-optimal designs. On the other hand, the optimum for the random design is exactly at $(-1, 1)$, which is one of the points sampled by the grid approximation. We generated a dozen such random designs and to our surprise, all of them engendered SPV surfaces with optima located in one of the corners. This indicates that the grid approximation would generally have little to no error for randomly generated designs. It is surprising that the randomly generated design engendered an SPV surface that was accurately scored using the grid approximation, but the discrepancy between the true maximum and the approximated maximum may be more pronounced for non-random, sub-optimal designs. It should be noted, however, that the SPV surface changes every time the outer optimizer proposes an update, and the relationship between approximation error and a design's proximity to G-optimality has yet to be explored in literature. We will address the relationship between optimizer update step and approximation error later in Chapter 4.

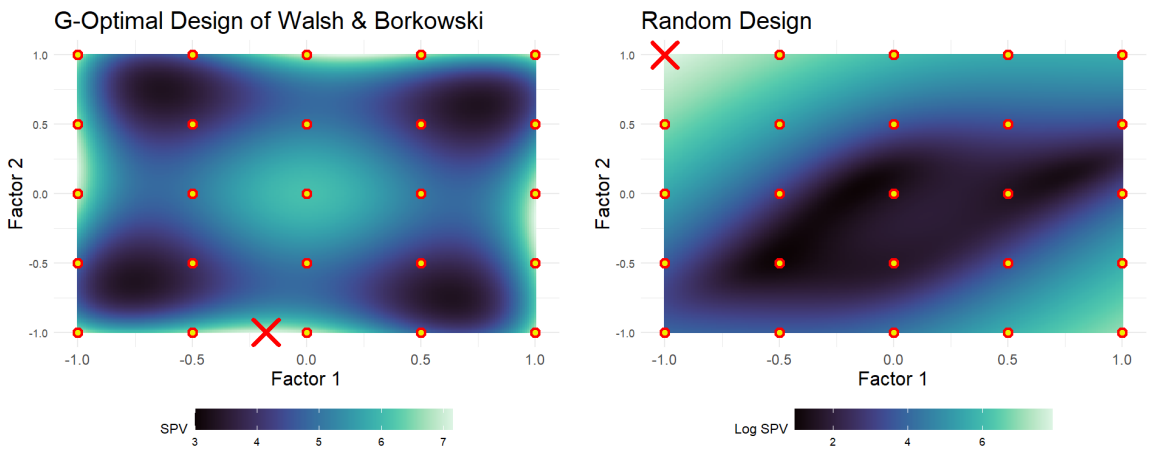

Figure 2.1: Side-by-side SPV-surface plots for the G-optimal design of [32] (left) and a completely random design (right). Locations that would be sampled by the grid approximation of [1, 32] are given as yellow circles outlined in red. The optimal design is visually representative of a G-optimal design after convergence of an optimizer, the random design would be a starting point used as the optimizer input.

2.5

`Gloptipoly` Usage in Matlab

The details of the `Gloptipoly` algorithm are too complex to spell out in this thesis, but the implementation in MATLAB makes using semidefinite relaxations to solve GPM straightforward. Syntactically, it parallels MATLAB's native support for symbolic programming and there are relatively few tuning parameters. To better understand how `Gloptipoly` works, we start with an example. Since the focus of this work surrounds the G-criteria – which corresponds to the maximum value on the SPV-surface – let's consider the best known design for $K = 2$ factors and $N = 6$ trials. The best-known design of [32] is given as

$$\mathbf{X}_{opt} = \begin{bmatrix} 0.17 & -0.21 \\ -0.51 & 1.00 \\ -0.86 & -0.97 \\ 1.00 & 0.83 \\ 1.00 & -1.00 \\ -1.00 & 0.54 \end{bmatrix}.$$

We define this matrix in MATLAB as shown in Listing 2.1. We then use the keyword `mpol`

from the `SeDuMi` add-on to define x_1 and x_2, the polynomial variables. A variable of type `mpol` is functionally similar to MATLAB's native symbolic variable representation, usually declared with the keyword `vars`. The code then defines the model vector and a corresponding vector of constraints, $\mathbf{x}_i \in [-1, 1]$, $i = 1, ..., K$. The design matrix $\mathbf{X}$ is then mapped to the model matrix $\mathbf{F}(\mathbf{X})$ using an expansion function `x2fx` from the Statistics and Machine Learning toolbox.

```
% Define the G-optimal matrix from Walsh & Borkowski (2022)
X = [0.17030087162924815 -0.21244490641568772;
    -0.5079431632696131 0.9999994648605898;
    -0.8557190575325034 -0.9691315280363174;
     0.9977983816447024 0.830995957569407;
     0.9999998764844229 -0.9999994400816095;
    -0.9999998664977112 0.5363924518991028];

% Declare variables, create model vector, build constraints
mpol x1 x2
var = [1 x1 x2 x1*x2 x1^2 x2^2];
K = [x1 <= 1, -1 <= x1, x2 <= 1, -1 <= x2];

% Expand design to model matrix
F = x2fx(X, 'quadratic');
```

Listing 2.1: MATLAB code required to compute the G efficiency of the optimal design matrix as provided by Walsh and Borkowski [32]

Now we have built all the required components to define a scalar polynomial and use `Gloptipoly` to find its global maximum. Listing 2.2 shows how to define the SPV polynomial with references to Listing 1.1. Printing `f` to the console gives us the polynomial we're interested in optimizing. The SPV-surface for this candidate design corresponds to

$$
\begin{aligned}
SPV^*(\mathbf{x}'|\mathbf{X}) &= 6.10 + 1.12x_1 - 1.01x_2 - 5.63x_1^2 + 0.63x_1x_2 - 6.32x_2^2 - 1.12x_1^3 \\
&\quad + 0.49x_1^2x_2 - 0.51x_1x_2^2 + 1.01x_2^3 + 7.56x_1^4 - 0.35x_1^3x_2 - 2.44x_1^2x_2^2. \\
&\quad + 0.21x_1x_2^3 + 8.25x_2^4. \\
G(\mathbf{X}) &= \max_{x_1,x_2} SPV^*(\mathbf{x}'|\mathbf{X}).
\end{aligned}
$$

Next we find the maximum value over $SPV^*(\mathbf{x}'|\mathbf{X})$ using `Gloptipoly` and `SeDuMi`, setting up the LMI hierarchy with `msdp` and finding the accumulation point of optimal solutions with `msol`. The third argument to `msdp` is the only hyperparameter-like input to `Gloptipoly`. This argument corresponds to the maximum order of LMI relaxations we want to use and the algorithm is not sensitive to its value in most cases.

```
% Define the polynomial
f = 6*var*inv(F.'*F)*var.';

% Search for optimum using \texttt{Gloptipoly}
P = msdp(max(f), K, 5);
[~, SPV] = msol(P);

% Extract the maximizers
x = double([x1 x2]);
```

Listing 2.2: MATLAB code to run `Gloptipoly` and compute the G-score of a candidate design. `f` is a polynomial representation of the scaled prediction variance (SPV) surface corresponding to the $K = 2$, $N = 6$ best-known design.

After running `Gloptipoly`, the `mpol` variables `x1` and `x2` now contain the optimal arguments. We convert them to type double and can easily extract values. For this candidate design, the optimizer is given as

$$
\mathbf{x}' = \begin{bmatrix} 0.09 & 1.00 \end{bmatrix}.
$$

The functionality of the `Gloptipoly` interface extends far beyond the example presented here and

interested readers should consult [7] for usage information and problem-specific instructions.

2.6

Conclusion

In this section we introduced the generalized problem of moments, the problem class to which computing the G-score belongs, and presented `Gloptipoly` as a tool that can perform the computation. We showed the $SPV(\mathbf{x}'|\mathbf{X})$ polynomial for the second-order model is a fourth-order semidefinite polynomial whose theoretical properties make it a good candidate to be optimized by `Gloptipoly`. We provided proof-of-concept by applying `Gloptipoly` to the best-known G-optimal designs published by [32]. Finally, we gave MATLAB code to use `SeDuMi` and `Gloptipoly` to score candidate designs according to the G-criteria. In the next section, we will use `Gloptipoly` as the objective function in a design search and assess its efficacy relative to state-of-the-art methods.

CHAPTER 3
FINDING G-OPTIMAL DESIGNS WITH THE NELDER-MEAD SIMPLEX ALGORITHM AND `GLOPTIPOLY`

3.1
The Design Search

Having demonstrated `Gloptipoly`'s effectiveness for scoring candidate designs, we now turn our attention to the design search. Mathematically, the problem is formulated as

$$\mathbf{X}^* = \underset{\mathbf{X}\in\chi^N}{\operatorname{argmin}} \max_{x'\in\chi} N\mathbf{f}'(\mathbf{x}')(\mathbf{F}'\mathbf{F})^{-\mathbf{1}}\mathbf{f}(\mathbf{x}')$$

where $\mathbf{X}^*$ corresponds to the G-optimal design. Procedure advocated by [4] involves scripting an algorithm that uses `Gloptipoly` as the loss function inside a CEXCH design search. The exchange method works well when the space of candidate designs is low-dimensional or when the objective function is computationally cheap to evaluate [19], so it would be fitting for something like a D-optimal design search because thousands of runs can be computed in a short time. By contrast, the method of semidefinite relaxations utilized by the `Gloptipoly` algorithm [15] involves the formulation and solution of multiple SDPs to score just one candidate design. So if $\mathbf{X}^*$ is $N \times K$, CEXCH calls `Gloptipoly` $N \times K \times P \times E$ times, where P is the number of passes through a candidate design before no changes are made and E is the number of times a chosen univariate optimizer calls the objective function on the X_{ij}^{th} element.

Several attempts were made to stay consistent with standard practice in the field and fit CEXCH to the G-opt search. Different flavors of the exchange algorithm were tested including the element-by-element exchange algorithm of [4] and a row-exchange algorithm introduced by [20]. Results from these searches were generally poor and took an impracticable amount of time for the number of design scenarios we wanted to cover, but for the purpose of comparison we will give a brief treatment to these algorithms in Chapter 5. In this chapter, we utilize a gradient-free multivariate optimizer to search the design space. We tried several outer-optimizers before settling on the Nelder-

Mead simplex algorithm.

The Nelder-Mead simplex algorithm is a multivariate optimization algorithm popular for its versatility. It's often used when the loss function is not known to be convex or differentiable, making it suitable for the min-max search required by G-optimal designs. The algorithm essentially constructs a simplex that "crawls" through the search space, beginning with a set of $N + 1$ points to form the simplex where N is the dimension of the parameter space. The objective function is then evaluated on all points in the set. Output values are ordered by quality and the centroid is calculated on the simplex formed by the best N points. The worst point in the set is then reflected through the simplex by finding a point equidistant from the centroid but in the opposite direction. Depending on the outcomes of reflection, the algorithm may expand, contract, or shrink the simplex to explore the parameter space efficiently. The process continues until a termination condition is met, and the algorithm returns the best point found as the estimated optimum of the objective function.

3.2

Implementation

The Nelder-Mead simplex algorithm implementation from MATLAB's optimization toolbox was used for the outer minimization search. In other words, in this section we use Nelder-Mead to search the space of candidate designs and propose updates. The routine is called with the `fmincon` function, which allows the inclusion of constraints. By default, the routine constructs initializes a simplex on the search space by adding 5% to the initial condition provided for each variable. It then searches the space as described in Section 3.1 and returns the found optimum. Now, the algorithm is designed to find optima for vector-valued functions, but the design search optimizes a function with a matrix-input. We can work around this issue by employing a matrix operator that takes columns of the matrix and stacking them, creating a vector of matrix entries that will just need to be converted into a matrix in the objective function. To illustrate the use of the `Vec` operator, begin with an arbitrary design matrix $\mathbf{X}$ and partition it by columns. The $N \times K$ design matrix is expressed as

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_K \end{bmatrix}.$$

The `Vec` operator will simply take each $\mathbf{x}_i$ and embed it in a column vector, moving from left to right across the original matrix. Then,

$$\text{Vec}\left(\mathbf{X}\right) = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_K \end{bmatrix}.$$

MATLAB has a convenient built-in function that makes the conversion easy, called `reshape`. This function takes as its first argument a matrix or array, and has a size parameter as its second argument. Example code is given in Listing 3.1 for the initialization of a random starting point and subsequent conversion to vector. Two arguments are given for size. The first size argument, `[]`, tells the function to adjust the size of the output to have as many rows as is necessary to make the data compatible with the second size argument, which specifies the number of columns. This code achieves the desired mapping from vector to matrix.

```
% Generate a random starting point with N rows and K columns
X = gen_mat(N, K);

% Call Vec on the matrix making it suitable for optimization
x_vec = reshape(X, [], 1);
```

Listing 3.1: MATLAB code to generate and reshape a matrix. The reshape function is given two size parameters, allowing it to dynamically change the shape of the output based on the dimension of the input data.

The optimal design search begins with a random starting point, so we built a function called `gen_mat` to generate random matrices with entries $x_{ij} \in [-1, 1]$ drawn uniformly. Pseudocode is given in Algorithm 1.

Algorithm 1 Generate Random Matrix

function GEN_MAT(num_trials, num_vars)
    $vals \leftarrow$ random values from $[-1, 1]$ with size num_trials $\times$ num_vars
    $X \leftarrow$ empty matrix of size num_trials $\times$ num_vars
    for $i \leftarrow 1$ to num_trials **do**
        for $j \leftarrow 1$ to num_vars **do**
            $X[i, j] \leftarrow vals[i \times j]$
        end for
    end for
     return X
end function

Next, the objective function needs to be modified to allow vector inputs while still formulating the correct SPV polynomial. If we pass in `x_vec` as the initial condition for the search, the found optimum will be a vector of optimal values needing conversion to a matrix for the operations in the SPV calculation to be conformable. After ensuring the data is formatted correctly, we need to formulate the SPV polynomial as in Listing 2.1, and call `Gloptipoly` to find the maximum SPV. A function was written to execute all these steps for any $N \geq p$ where p is the number of columns in the model matrix, and $1 \leq K \leq 5$. Code is presented in Listing 3.2.

```
% Compute G score for a candidate design using \texttt{Gloptipoly}
% ------------------------------------------------------------
function[SPV] = compute_g_vectorized(design, trials, num_var)

    % Convert vector to matrix
    X = reshape(design, trials, num_var);

    % Build x vector and constraints
    % var and K are defined as in Listing 2.1,
    % complete code is ommitted here for conciseness.

    % Evaluate G-score on matrix
    F = x2fx(X, 'quadratic');

    % Define the polynomial
    f = trials*var*inv(F.'*F)*var.';

    % Search for optimum using \texttt{Gloptipoly}
    P = msdp(max(f), K, 5);
    [~, SPV] = msol(P);

end
```

Listing 3.2: A function that used `Gloptipoly` and `SeDuMi` to find the max SPV for a candidate design. The function is defined with flexibility in mind, allowing a range of design scenarios to be studied.

Now that the objective function can handle vector inputs, it can be formulated as the inptut to an outer optimizer. Listing 3.3 provides the setup code needed to call `fmincon`.

```
 % Initial values
x0 = reshape(X, [], 1);

% Parameters for the optimizer that must be included but are
    unimportant
A = [];
b = [];
Aeq = [];
beq = [];

% Constraints
lb = -ones(N*K, 1);
ub = ones(N*K, 1);

 % Try fmincon
f = @(x)compute_g_vectorized(x, N, K);
[x_optimal, fval] = fmincon(f, x0, A, b, Aeq, beq, lb, ub, []);
```

Listing 3.3: Code used to call `fmincon` in MATLAB. Some of the inputs are superfluous, but they must be included for the function to excute. `A` and `b` would be used to formulate linear function constraints, but for this problem the only needed constraint is constant.

Becasuse Nelder-Mead is a local optimizer, we increase the chance of finding a good solution by calling the function multiple times with different starting points.

3.3

Saturated Cases and Infeasible Initial Values

The cannonical G-optimal design scenarios studied by [1] include cases where the model is saturated—that is, the number of parameters we need to estimate to fit a model is equal to the number of data points. In this thesis, we aim to use `Gloptipoly` to guide the search for each of these design scenarios, including the saturated cases. Saturated cases can be tricky to deal with because the space of eligible matrices is sparse. To illustrate how sparse the space of candidate designs is for saturated cases, consider the following example. Suppose while studying the $K = 1$, $N = 3$ design scenario, we generate a random 3×1 vector $\mathbf{x_0}$. This vector can be expanded to the 3×3 model matrix $\mathbf{F_0}$, and we need $\text{Rank}(\mathbf{F_0}) = 3$ in order for $(\mathbf{F_0}'\mathbf{F_0})^{-1}$ to exist. There is a very small, though nonzero probability, that with uniformly-drawn random entries in $\mathbf{x_0}$ one row of $\mathbf{F_0}$ is a linear combination of the others, but the probability is not zero. Now say we add one more trial so that $N = 4$. This case allows for one row to be a linear combination of the others, and because of this redundancy the probability of getting a starting point that results in a rank-deficient information matrix is lower when compared to the saturated case. In other words, when the experiment utilizes more trials there are fewer starting points that would halt the search from its outset, making the space of feasible starting points less sparse when compared to saturated design cases.

Standard practice to prevent errors when searching for optimal designs is to compute the determinant of the information matrix before passing it into the optimizer, precluding searches that begin with rank-deficient information matrices. If

$$\text{Det}(\mathbf{F}'\mathbf{F}) = 0$$

the information matrix cannot be inverted and shouldn't be used as a starting point. Because numerical approximations inevitably induce some error, we usually set a threshold like

$$\text{Det}(\mathbf{F}'\mathbf{F}) \leq \text{eps}^{(1/k)}$$

where k is a hyperparameter to be tuned and eps represents machine precision. This procedure works when N is strictly greater than p, but it doesn't work as well for the saturated cases. Using the condition number of the model information matrix was demonstrably more reliable for these cases, though there remains a tuning parameter in the calculation. When a randomly-generated matrix fails to meet the condition we set for invertibility, we simply generate new random matrix.

We then repeat until we find one that does meet the criterion. The reciprocal condition number was used in the final search to select feasible starting points for saturated design cases with k set to be the smallest value that allows the script to execute.

3.4

Results

Five-hundred iterations were run on the 21 design-scenarios studied by [1]. The best known G-optimal designs to date were found by [32] using the particle swarm in conjunction with a grid approximation to compute the G-score. Covering all cases on an 18-Core i9 workstation with a Nvidia 3080 GPU and 32gb ram took approximately seven days. A full week is a long time, but guaranteeing a globally optimal G-efficiency for each candidate design justifies the added cost [7]. Runs were performed in parallel and the design with the highest G-efficiency was chosen for presentation. Table 3.1 summarizes the efficiencies of our search method relative to previously published G-optimal designs. G-`Gloptipoly` performed as well or better when compared to the other methods studied in the literature on all $K = 1$ and $K = 2$ design scenarios, with the G-PSO method of [32] being the only noticeably better method. For the $K = 3$ case there was more spread—likely due to the increased dimension of the search space—but results indicated that `Gloptipoly` is a good choice to evaluate the objective function when design-space symmetry cannot be exploited.

Studying these canonical design cases highlights the difficulty associated with finding G-optimal designs for high-dimensional search spaces. For all $K = 1$ design scenarios, the designs found with `Gloptipoly` were on-par with state-of-the-art (SOA) optimization procedures; however, as the number of factors and the number of variables increases, relative efficiencies tend to decrease. Recall that the $K = 3$, $N = 16$ case engenders a 48-dimensional outer optimization problem with a 3-dimensional inner optimization, so the search space is necessarily vast. This is also reflected in the boxplots of Figures 3.1, 3.2, and 3.3. As the dimension of the search space increases the range of local optima increases as well, with designs for the one-factor case generally surpassing 95% efficiency and three-factor design cases finding some obvious local optima with relative efficiencies as low as 20%. Note that there may be ways to improve our search, for example using `Gloptipoly` to evaluate the objective function inside a meta-huristic optimizer like the particle-swarm of [32]. We will explore this approach further in Chapter 6.

Boxplots for all 500 runs per design-scenario are given to emphasize the prevalence of local optima. Naturally, lower-dimensional problems have fewer local optima and the algorithm generally

Table 3.1: Relative efficiencies of G-optimal designs for each algorithm relative to GA generated designs of [1]. From left to right, efficiencies for the PSO of [32], the $G(I_\lambda)$ procedure of [18], the coordinate-exchange of [22], and the Nelder-Mead + `Gloptipoly` approach studied in this work.

Design Scenario		Best Design Efficiency Relative to G-GA)			
K	N	G-PSO	$G(I_\lambda)$-CEXCH	G-CEXCH	G-`Gloptipoly`
	3	100.0	100.0	100.0	100.0
	4	100.0	96.2	98.7	100.0
	5	100.0	97.0	98.7	100.0
1	6	100.0	100.0	100.0	100.0
	7	100.0	98.8	99.7	100.0
	8	100.0	94.7	99.4	100.0
	9	100.0	100.0	89.4	100.0
	6	100.3	94.1	96.5	99.0
	7	100.1	95.5	97.9	98.6
	8	100.0	94.7	99.7	99.2
2	9	100.3	95.8	97.0	97.8
	10	101.7	93.2	97.5	100.3
	11	101.0	97.0	94.0	100.1
	12	103.9	95.1	101.2	100.3
	10	101.6	95.4	93.1	94.3
	11	104.2	96.9	92.9	97.9
	12	103.8	90.3	90.7	99.6
3	13	103.2	99.9	92.9	94.3
	14	100.5	100.0	87.6	92.0
	15	102.5	100.1	98.5	94.4
	16	108.1	100.2	100.1	97.4

finds designs with high G-efficiency, but as the dimension of the problem increases so does the number of local optima and hence the algorithm's propensity to entrapment.

Figure 3.1: Boxplots for 500 runs of Nelder-Mead on one-factor design scenarios. The best of these runs is consistently the G-optimal design for one-factor design scenarios.

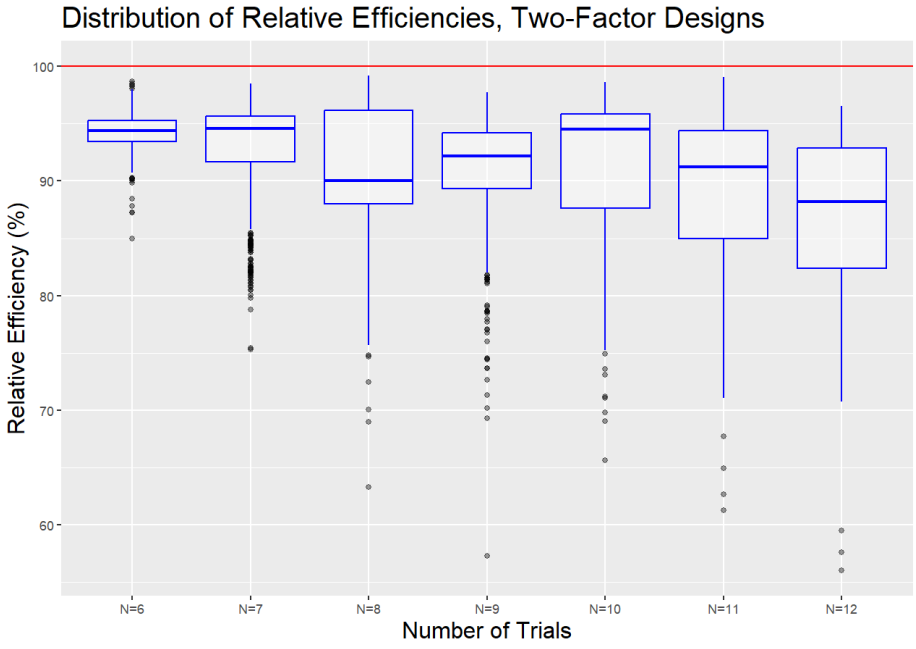

Figure 3.2: Boxlpots for two-factor settings. A two-factor design search results in significantly more spread when compared to the one-factor design search, likely due to an increase in the number of local optima.

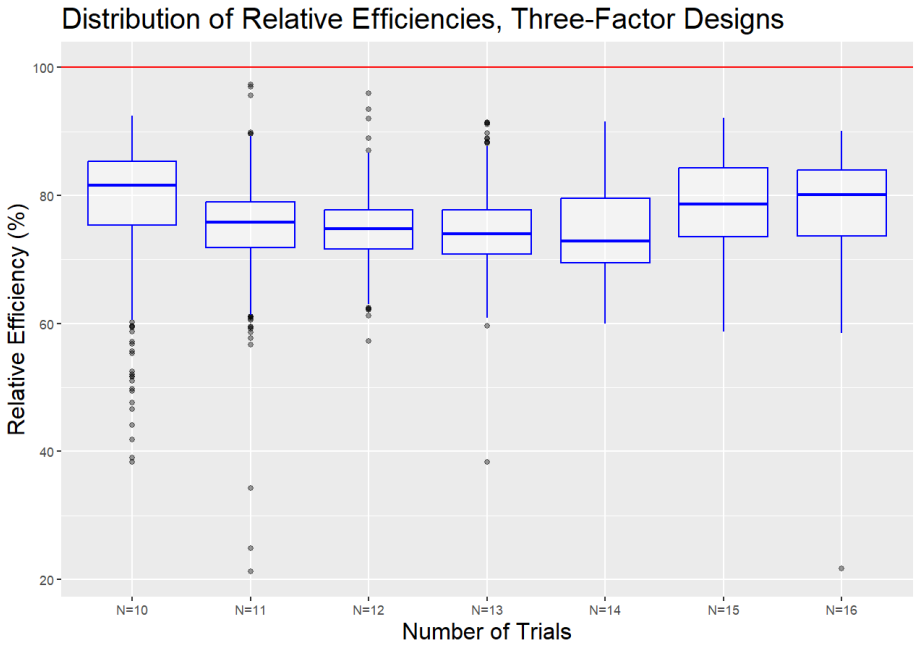

Figure 3.3: Boxplots for three-factor settings. Though not-quite SOA, the best designs have relative efficiencies above 90%.

3.5

Conclusion

In this section, we used `Gloptipoly` to guide the design search, embedding it in the objective function to score candidate designs. Implementation required a modification of the problem statement, transforming the $N \times K$ design matrix to an NK-dimensional column vector, ensuring conformability with MATLAB's optimizers. The design search process is outlined and includes working MATLAB code along with pseudocode for custom-built functions. Complexities associated with saturated design-scenarios were briefly discussed before presenting the results of five-hundred runs of the outer-optimizer per design-scenario. Absolute efficiencies for each case are visualized with box-plots and a discussion of the methods efficacy can be found in section 3.4. Though Nelder-Mead + `Gloptipoly` fails to match SOA, the designs we find are highly-efficient compared to other methods in the literature. In the next section, we will apply this method to design scenarios that have yet to be studied.

CHAPTER 4
HIGHER-ORDER MODELS AND APPROXIMATION ERROR

4.1
Introduction

In Section 1.5, we gave an overview of the methods used by researchers to compute the G-efficiency of a candidate design matrix. In Section 2.4, we re-scored the proposed G-optimal designs of [32]. Though the differences were only pronounced for some design scenarios, this cursory examination provides no insight into how much these calculations differ at each update proposed by the outer optimizer. To understand how the method used to evaluate the objective function affects the proposed G-optimal designs, we must further explore this relationship. We begin by extending the procedure to higher order models. Our literature search did not return any G-optimal designs for design-scenarios with higher-order interaction terms, so we begin by using the Nelder-Mead Simplex Algorithm introduced in Section 3.1 to find and propose highly G-efficient designs for first, second, and third order models with additional interaction coefficient estimates. We then compare the efficacy of the search for two objective function evaluation methods: `Gloptipoly` and the 5^k grid of [1].

In Section 4.2 we study the efficacy of Nelder-Mead and `Gloptipoly` to find G-optimal designs under two and three factor design settings with higher order interaction terms. We then extend the work to one, two, and three factor cubic and quartic response-surface models in Section 4.3. We also present a case-study examining the difference in computed max SPV for both the 5^K grid approximation of [1] and `Gloptipoly` at each update to the design matrix proposed by the Nelder-Mead Algorithm. Finally, we compare the results of Nelder-Mead with Glotipoly to the results of Nelder-Mead with the grid approximation to assess relative efficacy.

4.2

Models With Higher-Order Interaction Terms

When using experimental designs to study physical systems, practitioners often need to evaluate the significance of higher order interactions, particularly when studying biological and physical systems [24]. Here we address that need by proposing G-optimal designs for models with a second-order interaction term. The designs we find are highly G-efficient making them immediatly implementable; however, since there are no extant G-optimal designs for models with higher-order interaction terms in the literature they may also be used as a baseline for comparison in future work. We begin with the simplest case: finding G-optimal designs for a second-order model with higher-order interaction terms. Obviously, the inclusion of interaction terms implies a multivariate response-surface model, so we begin with a two-factor setting.

A practitioner may have theoretical justification to include any combination of higher-order interactions, but since this work is theoretical in nature we will pick one such model in two-factors:

$$f(x_1, x_2) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1^2 x_2 + \beta_6 x_1 x_2^2$$

where β_5 and β_6 are the coefficients for two new higher-order interaction terms. Because we are estimating two parameters on top of what is required for the full second-order model, the minimum number of experimental runs is 8 – two more than our RSM from Chapter 3. As addressed in Chapter 3, we need at least as many trials as we have parameter estimates to fit a linear model, so we will move forward using $N > 8$.

We ran 100 iterations of the Nelder-Mead Simplex Algorithm on $N = 9, 10$ using `Gloptipoly` to evaluate the objective function. Because computational feasibility is a typical consideration when searching for G-optimal designs [32, 18, 8], we performed 5 separate such searches and report the minimum, median, and maximum compute time for each design scenario at the end of Section 4.3 in Table 4.1. The first of these runs was used in reporting efficiencies. The median time for all 5 runs was 25.28 and 23.86 minutes for $N = 9$ and $N = 10$ respectively. It is interesting to note that the 10-trial setting required less time on average when compared to the 9-trial setting. Because the search for G-optimal designs is a minimax saddle-point optimization problem, it is difficult to conjecture a geometric or heuristic explanation for the decrease in computation time associated with a modest increase in problem complexity. The outer optimization for $K = 2$, $N = 9$ is an 18-variable optimization problem while for the $K = 2$, $N = 10$ setting it is a 20-variable optimization.

We generally expect more variables to require more compute time; however, this simple example contradicts that expectation.

Since this is the first piece of academic work studying G-optimal designs for higher order response surface models, we can't present efficiencies relative to the best-known designs. We can, however, utilize the general equivalence theorem outlined in Section 1.3 to calculate each proposed design's *absolute* G-efficiency. The general equivalence theorem gives p, the number of model parameters, as the lower bound on max SPV for any design-scenario, but since the actual max SPV for some design settings is much greater than p it is difficult to truly know how close to the global optimum we actually are without exhaustively testing a variety of outer-search algorithms. Nonetheless, in Figure 4.1 we provide boxplots of absolute efficiencies for the higher-order two-factor models discussed above. The best-design found for the 9-trial higher-order model was over 90% efficient and the best design for the 10-trial model was close to 85%. It is also interesting to note that the 10-trial search led to designs that are more consistently highly G-efficient as indicated by the dense IQR of the corresponding box plot. The intuitive explanation for this result is that the 10-trial setting has fewer local optima when compared to the 9-trial setting, which also likely explains the observed decrease in computation time.

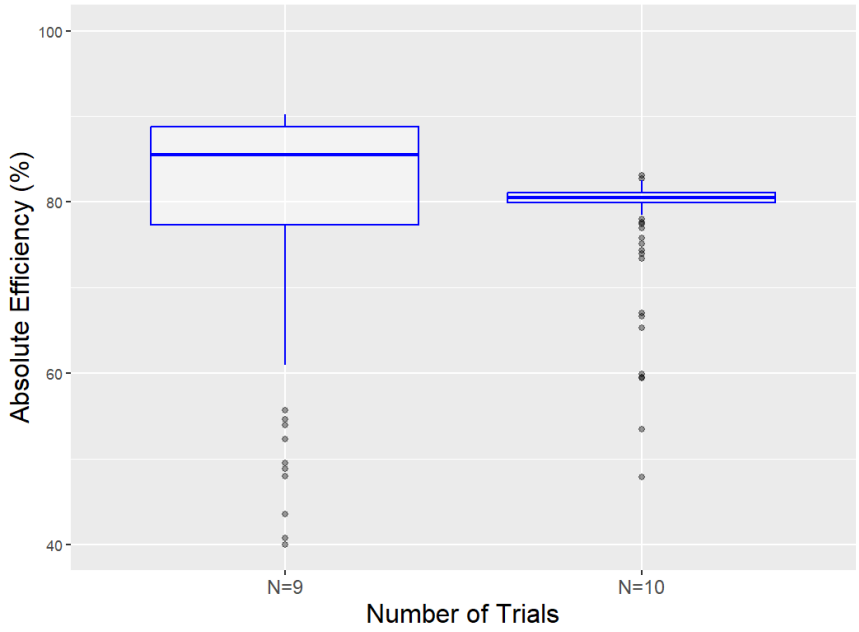

Figure 4.1: Boxplots of absolute efficiencies for two-factor response-surface models with 9 (left) and 10 (right) trials.

We also propose designs for two design settings utilizing the following three-factor model with

higher order interaction terms:

$$f(x_1, x_2, x_3) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \beta_4 x_1^2 + \beta_5 x_2^2 + \beta_6 x_3^2 + \beta_7 x_1 x_2^2 + \beta_8 x_1^2 x_2 + \beta_9 x_2^2 x_1.$$

This model had 13 parameters to estimate, so we chose to run our algorithm for $N = 14, 15$. The median times were 194.42 and 193.88 minutes for $N = 14$ and $N = 15$ respectively. Once again, though the number of variables we need to optimize for increases by 3 when moving from a 14-trial setting to a 15-trial setting, we see a counterintuative modest decrease in computation time. In this case, using the IQR density to conjecture a relationship about the number of local optima won't do since $N = 15$ seems to have more spread; however, the difference in compute time between the two settings is marginal compared to the difference we observed for the 2-factor model settings.

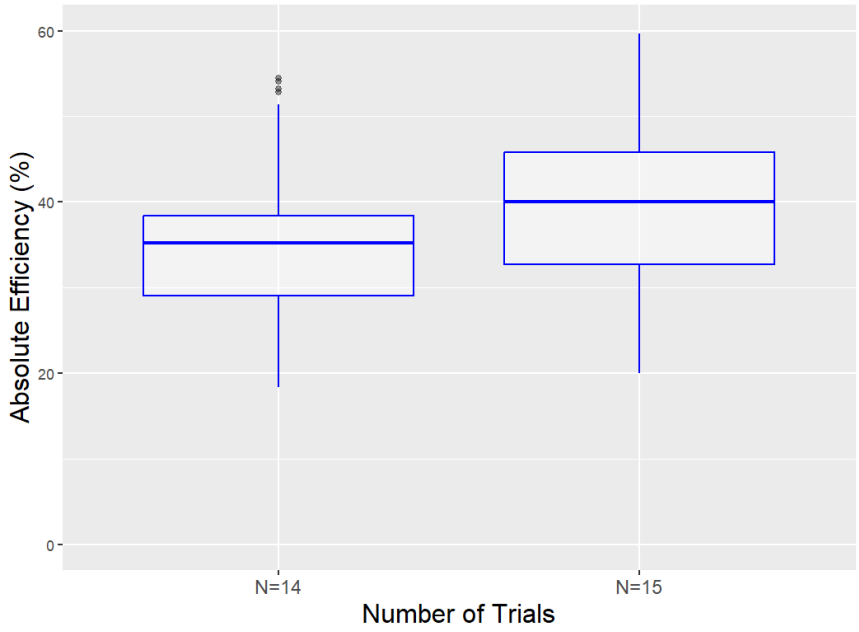

Figure 4.2: Boxplots of absolute efficiencies for three-factor response-surface models with 14 (left) and 15 (right) trials. Note that the absolute efficiencies are lower for the 3-factor model settings when compared to the two-factor settings. This is unsurprising since as the dimension of the problem increases, absolute efficiency generally decreases [32].

4.3

Cubic and Quartic Models

In addition to higher-order interaction terms, models may require cubic and sometimes quartic parameters. In this section we will find the first known G-optimal designs for RSM models satisfying this relationship. We begin here by studying the one-factor cubic model,

$$f(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3.$$

Boxplots for all 100 candidate G-optimal designs proposed by Nelder-Mead are presented in figure 4.3, and the corresponding run times are given in Table 4.1. For this study we considered the 5 and 6 trial settings, but the code is easily adapted to find G-optimal designs for any number of trials.

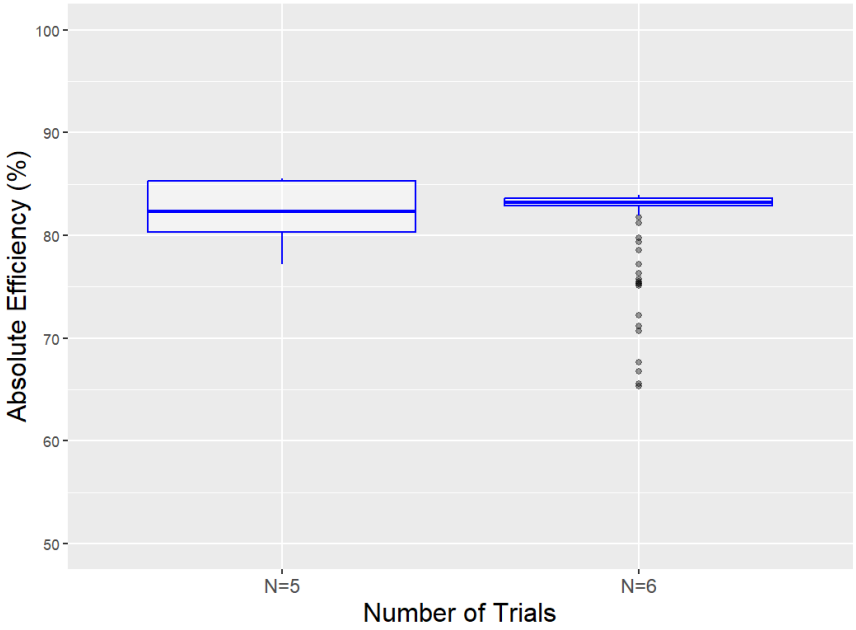

Figure 4.3: Boxplots of candidate optimal designs proposed by Nelder-Mead for the cubic RSM model in one factor. The 6-trial minimax problem seems to have a greater range of local optima when compared to the 5-trial problem.

In Figure 4.4 we visualize the SPV surfaces for optimal designs produced for these same one-factor settings. The grid approximation points of [1] are presented as yellow dots with red outlines. In both cases, the max SPV is achieved when $x = 1$, so the grid approximation works just as well as `Gloptipoly` for the univariate case.

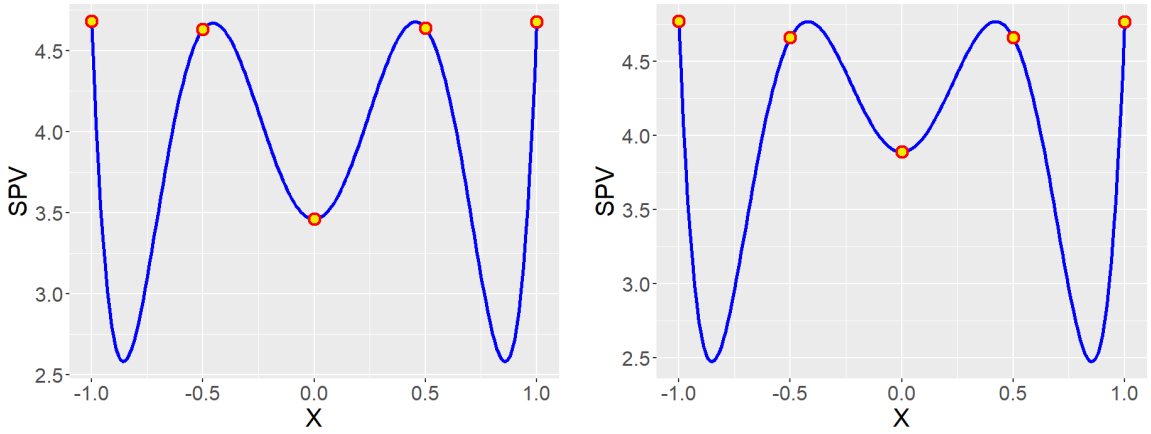

Figure 4.4: SPV-surfaces for the 5-trial (left) and 6-trial (right) cubic RSM models in one factor. These design scenarios engender geometrically similar SPV-surfaces.

The next set of cases we studied considered the two-factor cubic model with lower-order interactions. The two-factor cubic model had eight estimable parameters, so designs were generated for both the 9-trial and 10-trial design settings. The model took the form

$$f(x_1, x_2) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1^3 + \beta_6 x_2^3.$$

As with the one-factor cases, 100 iterations of the Nelder-Mead Algorithm were used, and five separate instances of the code were executed for benchmarking. There was a modest increase in compute time for the 10-trial setting when compared to the 9-trial setting. Figure 4.5 visualizes the SPV surfaces for $N = 9$ (left) and $N = 10$ (right). In contrast to yellow and red dots featured in Figure 4.4, the dots overlayed on the SPV-surfaces now represent proposed design points. That is, each dot on the surface is a row in the design matrix for our Nelder-Mead proposed G-optimal design. These represent the factor settings that an experimenter would use in order to minimize the worst-case-scenario prediction variance. As will be discussed in Chapter 6, there can be interesting geometry associated with design points, and optimal designs often sample the design space in a somewhat symmetric way. The 10-trial G-optimal design presented in Figure 4.5 places design points symmetrically about both the veritcal and horizontal axes.

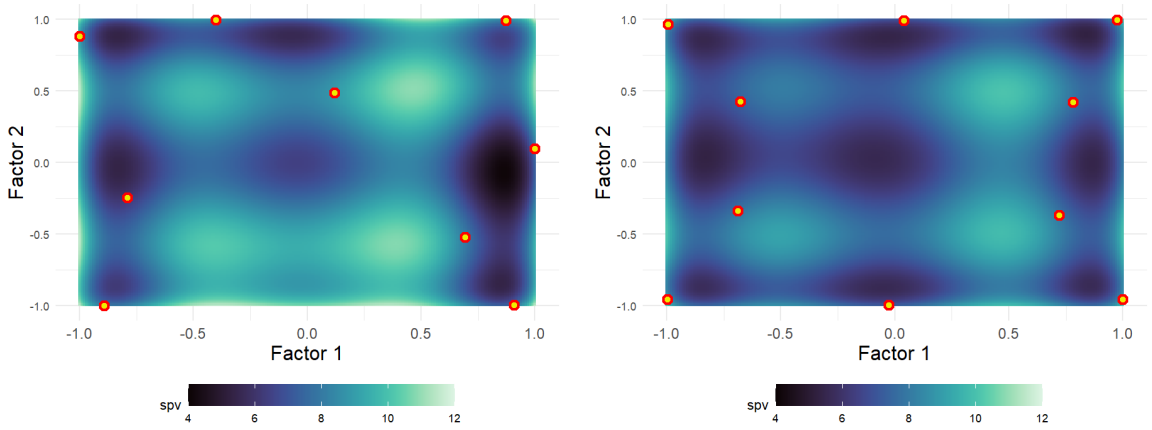

Figure 4.5: SPV surfaces for the cubic model in two-factors for 9 (left) and 10 (right) trials. Design points for the 10-factor setting are distributed about the surface symmetrically.

The next two factor setting we considered was the quartic model

$$f(x_1, x_2) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1^3 + \beta_6 x_2^3 + \beta_7 x_1^4 + \beta_8 x_2^4.$$

In Figure 4.6 we provide the SPV plot for the most G-efficient design proposed by our algorithm for $K = 11, 12$, respectively, with design-points superimposed.

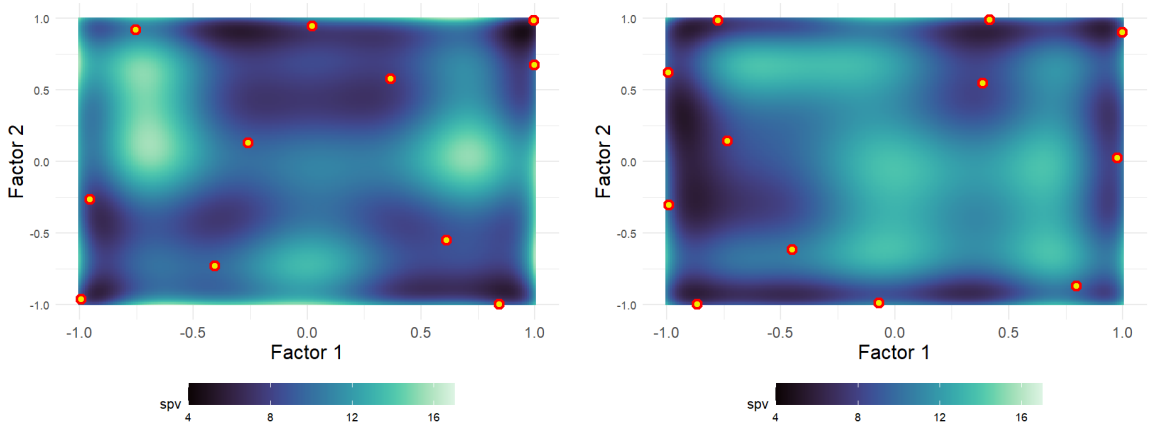

Figure 4.6: SPV surfaces for the quartic model in two-factors for 11 (left) and 12 (right) trials. There is no clear design-point symmetry for either of these surfaces.

Next, we considered experimental designs with three factors. We first explored a cubic model for $K = 14, 15$. This model includes first-order two-way interaction terms. Moving beyond the two-factor setting here requires four dimensions for visualization, one for each factor plus one for

the SPV value. This type of visualization can be obtained by plotting the SPV surface in three dimensions with color as the fourth, but doing so results in a figure that is difficult to present and interpret. Thus, to visually summarize the designs in three factors we use boxplots of the absolute efficiencies for each candidate optimal design proposed by Nelder-Mead for 100 algorithm runs. The three-factor model took the form

$$f(x_1, x_2, x_3) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \beta_4 x_1^2 + \beta_5 x_2^2 + \beta_6 x_3^2 + \beta_7 x_1^3 + \beta_8 x_2^3 + \beta_9 x_3^3.$$

A boxplot of absolute efficiencies is given in Figure 4.7. The two design settings engendered G-optimal designs with vastly different absolute efficiencies, but this is not unexpected.

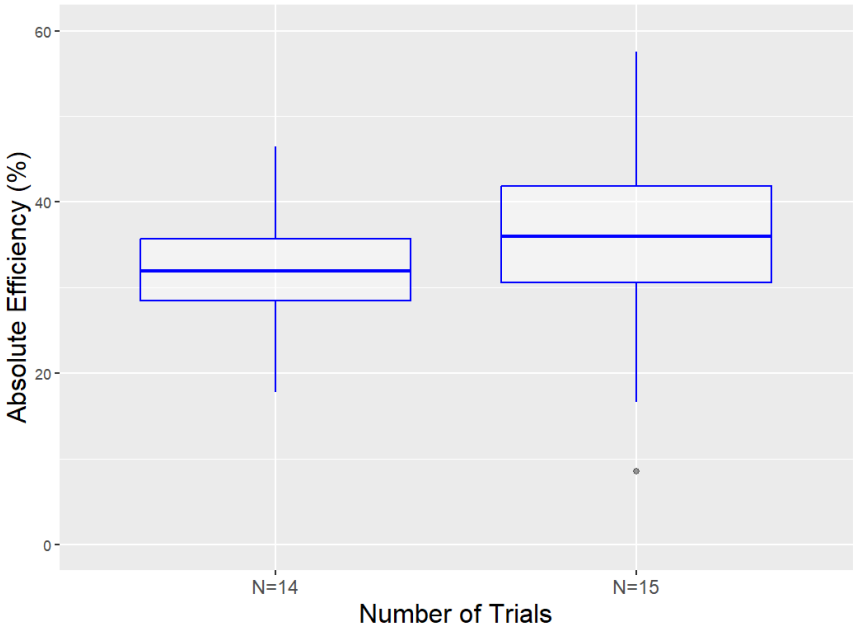

Figure 4.7: Boxplots of the absolute efficiencies for candidate G-optimal designs proposed by Nelder-Mead for the 14-trial (left) and 15-trial (right) cubic RSM models.

As with the two-factor settings, we next considered the quartic RSM model in three variables with the same number of trials as before. Interactions were not included so as to lessen computational burden. The chosen model took the form

$$f(x_1, x_2, x_3) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1^2 + \beta_5 x_2^2 + \beta_6 x_3^2 + \beta_7 x_1^3 + \beta_8 x_2^3 + \beta_9 x_3^3 + \beta_{10} x_1^4 + \beta_{11} x_2^4 + \beta_{12} x_3^4.$$

Side-by-side boxplots for the distribution of absolute efficiencies corresponding to each design scenario are given in Figure 4.8. The absolute efficiencies for the best designs in the candidate pools are relatively low compared to the other design scenarios; however, as mentioned earlier in this section and as observed by [32], we expect design efficiency to decrease as model complexity increases.

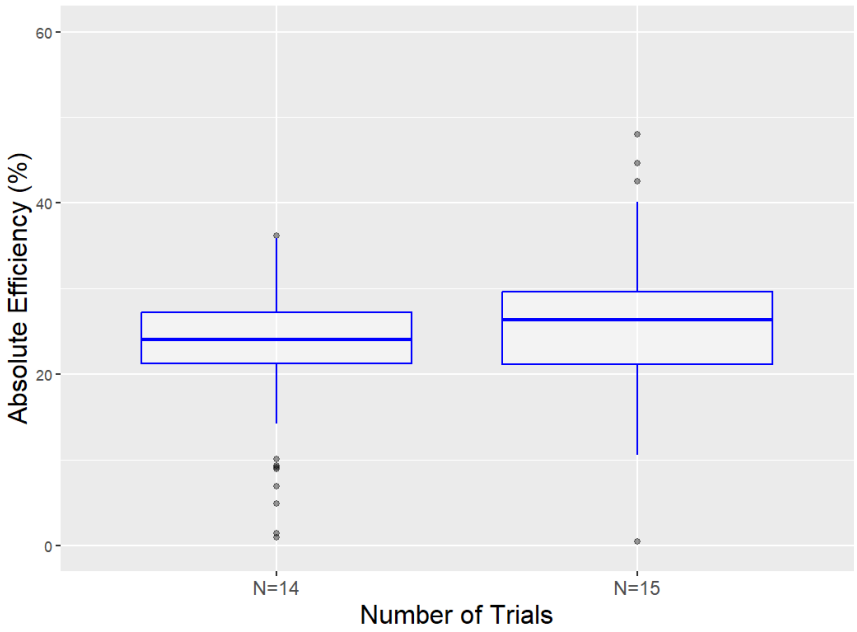

Figure 4.8: Boxplots of the absolute efficiencies for candidate G-optimal designs proposed by Nelder-Mead for the 14-trial (left) and 15-trial (right) quartic RSM models.

We end this section with benchmarking statistics given in Table 4.1, which provides the minimum, median, and maximum compute time in minutes for each design scenario. The 3-factor quartic models were omitted due to the computation time required to repeat 100-runs of Nelder-Mead five times. As expected computation time generally increases as model complexity increases, though the two-factor quartic searches actually took less time than the two-factor cubic models. This could be because the quartic models did not include an interaction term while the cubic models did.

K	N	Model	Min	Median	Max
1	5	Cubic	2.30	2.46	3.15
1	6	Cubic	4.08	4.11	4.12
2	9	Higher Order	25.05	25.28	25.84
2	10	Higher Order	23.78	23.86	25.13
2	9	Cubic	27.04	27.74	27.95
2	10	Cubic	28.64	28.67	29.00
2	11	Quartic	21.49	21.77	22.03
2	12	Quartic	24.54	24.56	24.84
3	14	Higher Order	192.92	194.42	194.68
3	15	Higher Order	192.72	193.88	194.68

Table 4.1: Table of run-times for selected design-scenarios. $K = 3$ quartic cases were excluded due to the computational burden required to repeat the search 5 times.

4.4

A case study on Approximation Error

We end this section with a brief exploration of the approximation error induced by the 5^k grid approach of [32, 1]. By using a dense grid approximation in Chapter 2, we were able to determine that `Gloptipoly` is a more accurate scoring mechanism when compared to the approximation scheme. In this section we make an effort to determine how the grid approximation of max SPV compares to the computed true optima calculated by `Gloptipoly`. The following section digs deeper into approximation error, computing the discrepancy between the two scoring algorithms at each update step of Nelder-Mead.

To begin this study, we collected data on the design matrix at every proposed update from the outer optimizer. These data included the design matrix at each Nelder-Mead iteration, the `Gloptipoly` computed max SPV, and the 5^k grid max SPV. As in Chapter 3, the full second-order model was used. We considered the design scenario with 2 factors and 10 trials. We ran 100 separate instances of Nelder-Mead, resulting in 100 different proposed candidate optimal designs. Figure 4.9 presents a boxplot describing of the number of iterations it took for Nelder-Mead to converge for each of the 100 runs. On average, Nelder-Mead takes 85.5 iterations to find a candidate G-optimal design, with a minimum of 21 runs and a maximum of 137. The wide range of iterations required for convergence hints at the complexity of the G-optimal problem.

Next we examined the relationship between update step and the absolute error between `Gloptipoly` and the grid approximation. Designs from the first run of Nelder-Mead were used, and max SPV was computed using `Gloptipoly` and the grid approximation. `Gloptipoly` was used to guide the search as the inner optimizer, but we performed post-hoc scoring of the designs at each

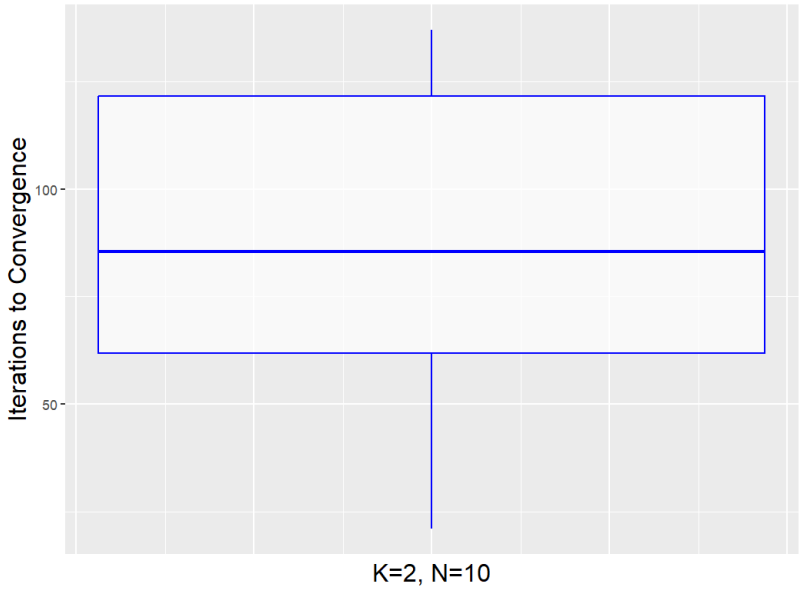

Figure 4.9: Boxplot of iterations to convergence for 100-runs of Nelder-Mead. There are some runs where the outer optimizer is quickly trapped by local optima and others where it searches for over 100 iterations before converging.

proposed Nelder-Mead update. The absolute value of the difference between the two scores was then calculated and plotted in Figure 4.10.

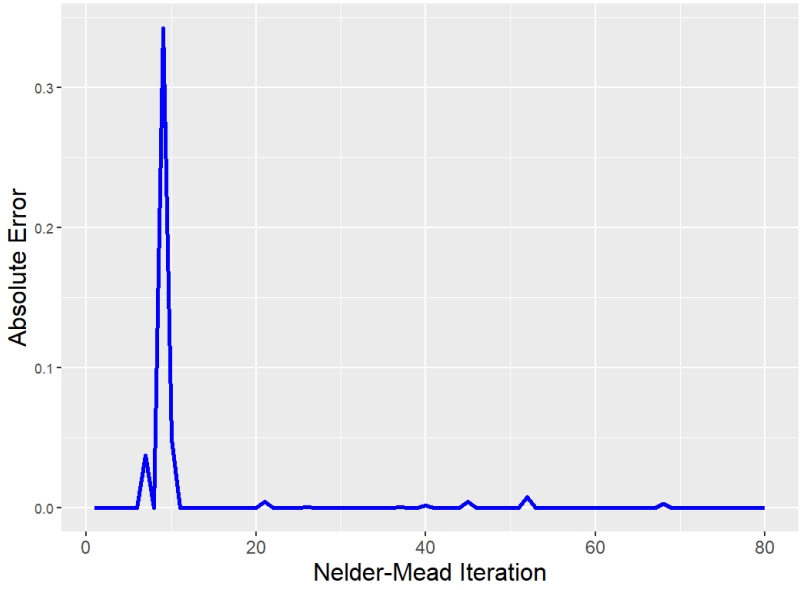

Figure 4.10: Line plot visualizing the difference between max SPV as computed by `Gloptipoly` and max SPV as computed by the grid approximation for the first run of the algorithm. The difference is generally quite small.

Because this is just one trial, more exploration is needed to determine if there exists a significant relationship between optimizer update step and approximation error. To assess the magnitude of approximation error, a boxplot of the average error across all proposed updates by Nelder-Mead for all 100 runs was produced. Outliers severely skewed the data, so absolute error is given on a log scale. This plot is given in Figure 4.11

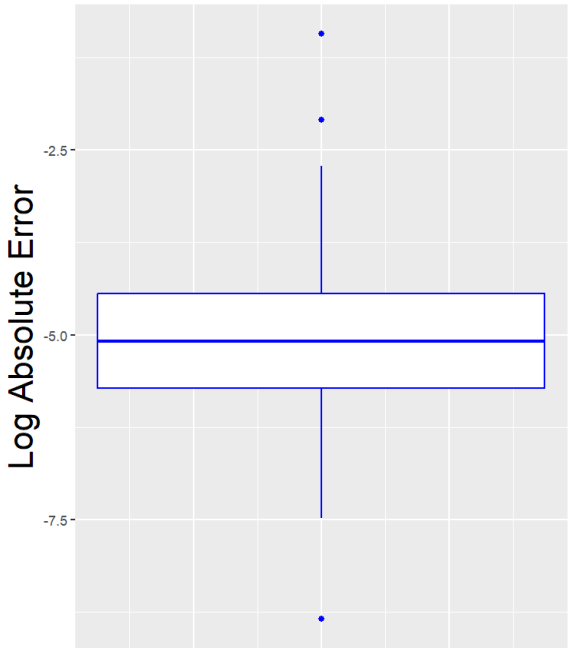

Figure 4.11: Boxplot of logged average approximation error. As we saw in Figure 4.10, most runs had fairly low approximation error. Some outliers had more significant error.

Further exploring the relationship between the iteration of Nelder-Mead and absolute error, we took all 100 runs of the algorithm and computed the average error at each step. As discussed above, some runs of the Nelder-Mead Algorithm had as few as 27 proposed updates until convergence while others took over 100 to find a local optimum. Data of varying lengths present computational difficulties, but to overcome this challenge the average of the i^{th} observation is taken to be the average of all runs where the i^{th} observation existed. Algorithm 2 outlines this procedure in more detail.

In Figure 4.12 we visualize the average error across all runs as returned by Algorithm 2. As the algorithm gets closer to convergence we see the absolute approximation error decrease. This could be due to the symmetry engendered by the second-order RSM model, or it could simply be that the magnitude of max SPV decreases significantly as Nelder-Mead proposes updates that flatten the

Algorithm 2 Calculate Average Values

```
max_length ← max(length(vec) for each vector in the list)
average_values ← numeric vector of length max_length
for i ← 1 to max_length do
    values ← numeric vector of length 100
    for j ← 1 to 100 do
        if i ≤ length(vec_j) then
            values[j] ← vec_j[i]
        else
            values[j] ← NA                ▷ If the ith observation doesn't exist in vector vec_j
        end if
    end for
    average_values[i] ← mean(values, na.rm = TRUE)
end for
```

SPV surface and bring all values closer to zero. Future researchers may consider producing plots paralleling this one but for a selection of response-surface models.

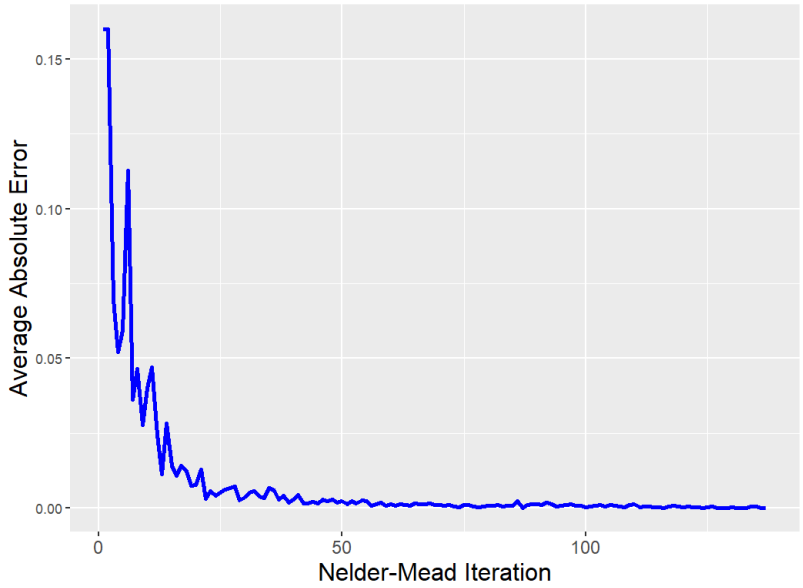

Figure 4.12: Line plot visualizing average approximation error at each update step across 100 runs of Nelder-Mead. The error decreases dramatically as the algorithm nears convergence.

4.5

Conclusion

In this section, we applied the `Gloptipoly` and Nelder-Mead pairing to a class of models with higher-order interaction terms. We then applied our algorithm to models with cubic and quadratic effects, followed by a brief case-study on approximation error to determine how big the discrepancy is

between the grid approximation and the method of semidefinite relaxations. While the two methods do return different values for the max SPV of a given surface, the magnitude of error is usually not very large and it decreases with each optimizer iteration. As such, the grid approximation is shown to be a good tool for estimating max SPV, but if experimental conditions necessitated the most highly G-efficient design possible, the practitioner ought to use `Gloptipoly`, at least for fine-tuning the design.

The studies in this chapter were conducted using the full, second-order model. Future research should consider performing similar case-studies, where approximation error is computed for non-second-order models like those introduced earlier in this chapter. As we saw in Chapter 3 and throughout this chapter, we tend to observe an increase in the number local optima on the SPV-surface as the model complexity increases. As such, it is likely that during the outer-optimization process there are intermediate, sub-optimal designs whose SPV-surfaces don't have a global maximum that is well-approximated by a 5^k grid-search approach. These designs may lead the outer-optimizer astray and result in sub-optimal designs.

We also noted a steady drop in absolute efficiencies for higher-order models. That is, when transitioning from second to third to fourth order response surface models, we note a decrease in absolute efficiency. This could be due to an increase in problem complexity without a commensurate increase in the number of outer-optimizer runs, but we also know from past research that absolute efficiency decreases as model complexity increases. Nonetheless, the relationship between complexity and efficiency would be an interesting problem to explore in future research.

CHAPTER 5
POINT EXCHANGES

5.1
Introduction

Despite recent success of multivariate meta-heuristic optimizers, the coordinate exchange (CEXCH) algorithm of [18] remains the most popular optimization routine to find optimal experimental designs [32]. We speculate that the reason for this is cultural rather than practical, as PSO is demonstrably the most suitable algorithm for this problem. Notwithstanding the success of PSO, researchers in the field of optimal design remain attatched to point-exchange algorithms, so in this section we make an effort to implement intelligently-guided point exchange algorithms.

The dominant paradigm involves discritizing the space of design points via a candidate set of size n. The coordinate exchange algorithm considers individual entries in the design matrix, swapping x_{ij} for a value in the candidate set if and only if that value results in an improvement to the design. One pass through the matrix requires $N \times K \times n$ objective function evaluations, so for a computationally expensive objective function like the G-criteria this procedure is asinine. An alternative exchange-algorithm considers exchanging rows of the design-matrix rather than individual entries. This procedure is often called the point-exchange (PEXCH) algorithm because proposed updates to the design replace design-points instead of individual coordinates. One pass of PEXCH requires $N \times n$ objective function evaluations, though the candidate set is often bigger than the candidate set for CEXCH.

5.2
Standard Point-Exchange

In this section we implement standard point-exchange, but instead of a uniformly spaced candidate-set we harness information extracted from the `Gloptipoly` algorithm to propose intelligent updates. As discussed in Section 4.4, with Nelder-Mead as the outer-optimizer it takes an average

of 85.50 iterations to converge. To illustrate the usefulness of this procedure, consider the SPV surface for a candidate optimal design at update step t vs. update step $t+1$. Especially towards the end of the design-search process, these SPV-surfaces are unlikely to undergo dramatic changes between iterations. To better understand this relationship, we arbitrarily chose to investigate the 36^{th} run of Nelder-Mead for $K = 2$, $N = 10$. This run took exactly 50 iterations to converge, resulting in a max SPV of 8.2. Figure 5.1 juxtaposes the SPV surfaces for update step 32 and 33, highlighting the fact that there has been very little change to the surface. Though the design points have moved slightly engendering midly varied SPV surfaces, it is visually challenging to distinguish any differences between the two plots. Accordingly, the max SPV difference is only 0.03, indicating that information at one update step is largely preserved to the next. The preservation of information provides theoretical justification to consider a point-exchange algorithm.

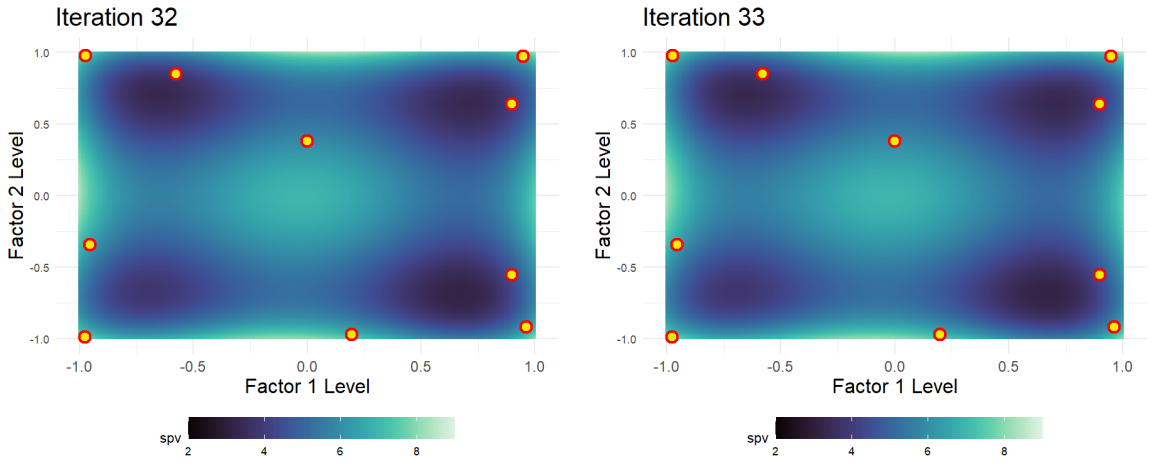

Figure 5.1: SPV-surfaces and superimposed design-points for the 36th run of Nelder-Mead on $K = 2$, $N = 10$. Visually, these designs are very similar.

NM Trial 36, Update 32

$$\begin{bmatrix} -0.003 & 0.384 \\ 0.959 & -0.918 \\ 0.899 & 0.644 \\ -0.956 & -0.344 \\ -0.579 & 0.852 \\ 0.946 & 0.974 \\ -0.979 & -0.986 \\ -0.977 & 0.981 \\ 0.896 & -0.551 \\ 0.195 & -0.969 \end{bmatrix}$$

$\xrightarrow{\text{One pass of Nelder-Mead (NM)}}$

NM Trial 36, Update 33

$$\begin{bmatrix} \mathbf{-0.001} & \mathbf{0.382} \\ \mathbf{0.960} & -0.918 \\ \mathbf{0.898} & \mathbf{0.642} \\ -0.956 & \mathbf{-0.341} \\ \mathbf{-0.581} & \mathbf{0.854} \\ 0.946 & 0.975 \\ -0.979 & -0.986 \\ -0.977 & 0.981 \\ \mathbf{0.897} & \mathbf{-0.554} \\ 0.195 & -0.969 \end{bmatrix}$$

Since the G-criterion is to minimize the wost-case-scenario prediction variance, we should use the optimizer (i.e. the $\mathbf{x}' \in [-1, 1]^K$ that describes the location of the maximum value on the SPV surface) to update the design. Heuristically, if we want to minimize worst-case-scenario prediction variance we should sample points that give us the most information about regions of high variability, and the optimizer theoretically captures the desired information. So in this section we implement a version of point-exchange where the candidate set consists only of the optimizer. We first take the current design and compute the global maximizer on the SPV surface using Gloptioply. We then build a list of $N + 1$ matrices and their corresponding G-scores, picking the design with the highest efficiency. Then, we repeat this procedure until no improvements are made to the design, resulting in a candidate optimal design. Pseudocode for a single run of the algorithm is given in Algorithm 3. To fully replicate our results, one need only embed this code in a 'for' loop and repeat as many times as desired.

The procedure was repeated 1000 times for the second-order response surface model on $K = 2, 3$, $N = 8, 12$ respectively. In Figure 5.2 we juxtapose the proposed optimal design from the Nelder-Mead optimization algorithm with the proposed optimal design from standard point exchange for $K = 2$, $N = 8$. Geometrically, the surfaces are similar to one another, and the design-points are placed in similar locations. The Nelder-Mead optimizer still produced a more efficient design than the point-exchange, with a relative efficiency of 99.2% vs 98.5% when compared to the design of [32]. The point-exchange still did, however, still perform surprisingly well on this case, especially considering how small the candidate set was.

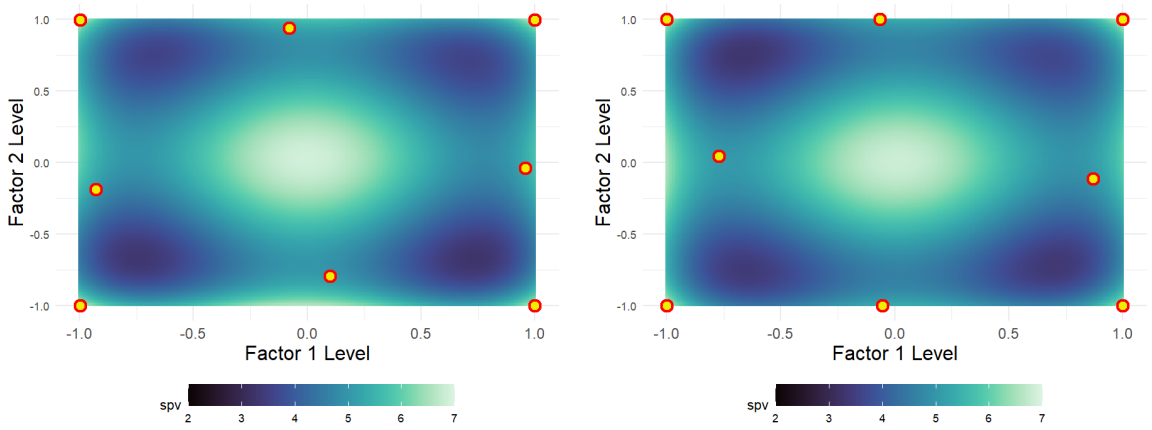

Figure 5.2: SPV surfaces corresponding to the Nelder-Mead proposed optimal design (left) and PEXCH proposed optimal design (right) for $K = 2$, $N = 8$.

Algorithm 3 `Gloptipoly`-Informed Point Exchange

```
while delta_G > tol do
    # Get max SPV for previous iteration
    [g_curr, optimizer] ← compute_g(design)
    # This is the optimizer
    x_new ← double(optimizer)
    # Append the optimizer to the matrix as a new row
    design_new ← [design; x_new.']
    # To store the max SPVS for the resultant matrices
    spvs ← zeros(1,num_rows)
    # Iteratively delete rows and re-score the matrix
    for j ← 1 to num_rows do
        # Remove a row from the new design matrix
        A ← design_new
        A(j, :) ← []
        # Compute the new g-efficiency
        spvs(j) ← compute_g(A)
    end for
    # Find the minimum max SPV
    min_spv ← min(spvs)
    min_index ← find(spvs == min_spv, 1)
    # Update the new design to be the best from the previous list
    design_new(min_index, :) ← []
    # Calculate the difference in old vs.  new design
    delta_G ← abs(compute_g(design_new) - compute_g(design))
    design ← design_new
end while
```

5.3

Modified Point-Exchange

As has been discussed at length, `Gloptipoly` is a computationally expensive algorithm. Every time the function is called, a hierarchy of SDPs must be formulated and solved, so it's important to build outer-search algorithms that can traverse the space of candidate designs with minimal objective function evaluations. In this section, we reduce the number of function evaluations by considering an update to the extant design each time `Gloptipoly` is called. Rather than creating a set of $N+1$ candidate designs and choosing the best at each iteration, we consider an update every time a design-point is swapped with the old design's optimizer. Algorithm 4 spells out the code in more detail. This algorithm continues to loop through the design until a certain tolerance is met between the max SPV at the previous iteration and the max SPV at the current iteration.

In Figure 5.3 we juxtapose the G-optimal designs for the two point-exchange methods. Design points are superimposed on corresponding SPV-surfaces. The designs have similar geometry and correspondingly similar G-efficiencies. Interestingly, the modified PEXCH engenders an SPV-surface whose maximum value is located at $(-1,-1)$ with a corresponding SPV of 6.98, while the standard PEXCH achieves its maximum of 6.92 at $(1,1)$. Additionally, the average SPV for both designs is about 4.65, so we would consider these designs to be of similar quality according to the I-criteria as well [8].

In Figure 5.4 we present side-by-side boxplots of the distribution canidate optimal designs for three algorithms: the two point-exchange methods introduced in this chapter and the Nelder-Mead algorithm proposed in Chapter 3. The efficiencies we present are calculated relative to the best known design of [32]. In Figure 5.5, we present boxplots of relative efficiencies for $K = 3$, $N = 12$. In both cases, Nelder-Mead still emerges as the best choice. We suppose here that restricting the candidate set to just the optimzer from the previous proposed optimal design may not provide the flexibility required to effectively search the space of candidate designs. This is not to say that this methodology is without use however, as it could provide very efficient starting points for a meta-heuristic optimizer like PSO. In other words, the top k designs from a run of PEXCH could serve as the starting points for a run of PSO. This could drastically reduce the number of iterations until convergence while simultaneously providing information to the optimizer that could result in a better design.

Algorithm 4 Modified `Gloptipoly`-Informed Point Exchange

while delta_G > tol **do**

`% Store last iteration's design`
old_design ← design;

`% Iteratively delete and re-score the matrix`
for $j \leftarrow 1$ to size(design, 1) **do**

`% Get max SPV for the previous iteration`
[spv_curr, optimizer] ← compute_g(design);

`% This is the optimizer`
x_new ← double(optimizer);

`% Replace row` j `in the design matrix`
proposed_design(j, :) ← x_new;

`% Compute max SPV of resultant design`
spv_new ← compute_g(proposed_design);

`% If resultant design is better, keep it!`
if spv_new < spv_curr **then**
design ← proposed_design;
else
`% Reset the proposed design if it's not better`
proposed_design ← design;
end if
end for

`% Re-calculate` δ_G `to determine termination`
delta_G ← abs(compute_g(old_design) − compute_g(design));
end while

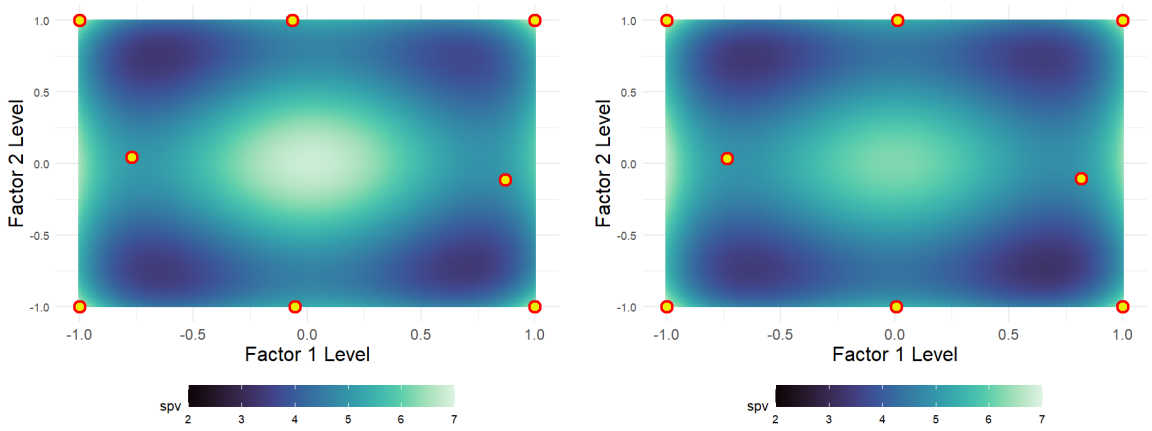

Figure 5.3: Comparison of SPV-surfaces for the two point-exchange algorithms tested in this chapter. We present standard PEXCH (left) as outlined in Algorithm 3 and our modified PEXCH (right) as outlined in Algorithm 4. The two methods find designs with comparable efficiency and the distribution of points is nearly identical.

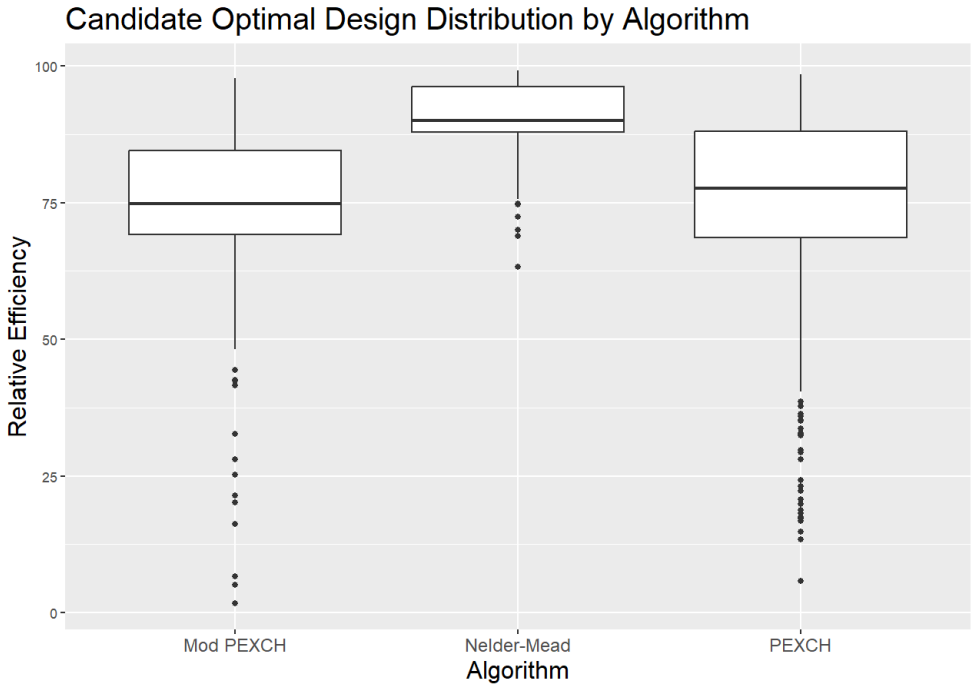

Figure 5.4: Boxplots for the distribution of designs proposed by three methods studied thus far in the text. For $K = 2$, $N = 8$, the two point-exchange algorithms have comparable performance. Nelder-Mead, however, is still the clear front-runner.

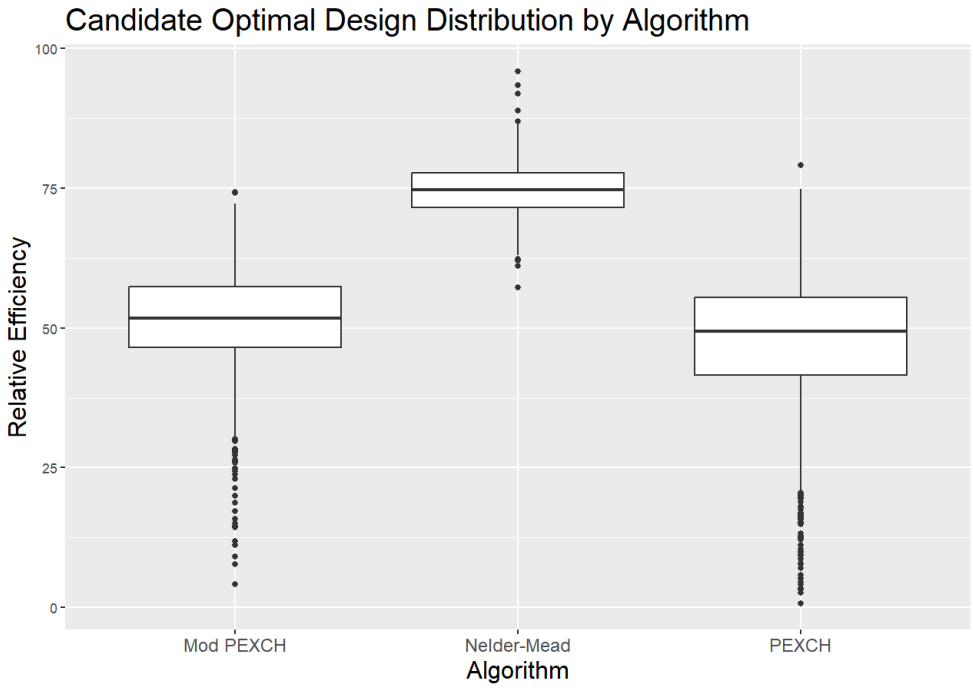

Figure 5.5: Boxplots for the distribution of three factor designs. For $K = 3$, $N = 12$, the modified point-exchange algorithm finds better designs on average, but the original point exchange finds the most G-efficient design. Nelder Mead is once again, the best option.

5.4

Conclusion

In this section we introduced two point-exchange algorithms: one that considers an update after generating a candidate list of $N + 1$ designs and another that considers an update every time a design point is swapped. Though there are historical reasons for implementing point-exchange algorithms, their performance falls short of modern optimizations routines like Nelder-Mead and PSO. Point-exchanges need not be disregarded entirely, as future work may consider using them to finding starting points for an outer optimizer. One would find a highly G-efficient design using PEXCH and use the proposed design points as the initial condition for a more flexible outer optimizer. Using input designs with theoretical structural similarities to output designs has been tried on this problem before by [8] to reduce computational burden. This section provides proof of concept to emulate their procedure, but with G-efficient designs from a cheap optimizer as the starting point rather than known I-optimal designs.

CHAPTER 6
EXTENSION TO STATE OF THE ART

6.1
Introduction

In this section we extend the work done in the previous five chapters by harnessing the precision of `Gloptipoly` with the intelligence of the particle swarm (PSO). Though the Nelder-Mead Algorithm was relatively successful, it failed to produce results on-par with the best-known designs of [32] in some cases. In this section we revisit the use of PSO, restricting ourselves to far fewer runs than is conventional in the field to make computation feasible. This work is motivated by two key factors. First, `Gloptipoly` is a guaranteed global optimizer when assumptions are met [6, 7] and was shown in Chapter 2 to provide more accurate scores than the grid search. Second, PSO offers the flexibility required to effectively explore large search spaces and is the current state-of-the art method for finding G-optimal designs [32]. The precision of `Gloptipoly` for evaluating the minimax objective function provides the cleanest signal to the outer optimizer to-date, and the ability of the particle swarm to consider multiple designs simultaneously provides the most thorough search of any outer-optimizer tested in this thesis. Testing this procedure on two and three factor design settings, we find that the intelligence of the particle swarm optimizer is sufficient to overcome limited trials. In doing so we produce the best-known G-optimal designs to-date for some design settings.

6.2
Particle Swarm Optimization

PSO belongs to the class of meta-heuristic optimizers, which means it navigates the space of candidate optima through trial and error. The algorithm was first introduced by James Kennedy and Russell Eberhart in 1995 [10]. It is said to have been inspired by the behavior of natural swarms, such as flocks of birds or schools of fish. Since its introduction, PSO has been applied

to a wide range of application spaces from economics to industrial engineering to optimal design of experiments. With over 80,000 citations, it has been an impactful contribution to the field of meta-heuristic optimization.

The power of the particle swarm comes from the ability of particles to "communicate" with one another, and the simultaneous ability of those same particles to move through different regions of the search space and relay update information to the swarm. Each particle represents a potential solution to the optimization problem and holds a weighted combination of individual and shared memory. The individual memory represents a particle's best-known-position (in our case, the design with minimum max SPV) and shared memory represents the best known position among all particles in the swarm. By leveraging collective knowledge, the particle swarm is able to search the space of candidate designs very effectively. We believe PSO is particularly well-suited to the G-optimal design problem because it is robust to entrapment in local optima. If the wide range of candidate designs proposed by Nelder-Mead and the PEXCH algorithms are any indication, this problem is replete with local optima. PSO aims to overcome the problem of local optima by using information from the best of the proposed designs to inform updates for each particle.

Now, [4] suggest performing several thousand runs of a greedy outer-optimizer like PEXCH to overcome entrapment in local optima. Since PSO is not a greedy algorithm and instead explores the solution space in a collectivist fashion, guided by the interactions of particles with each other, we believe we can find highly G-efficient designs with far fewer runs. In this section we allow just 20 runs of the outer optimizer and still get results on-par with [32], suggesting that when the outer-optimizer is well-suited to the G-optimal problem, we need far fewer runs than when we use greedy, naive optimizers like CEXCH or PEXCH.

6.3

Application of PSO to the Search for G-Optimal Designs

PSO was applied to the $K = 2, 3$ cases of [1]. $K = 1$ cases were omitted because most outer-optimizers are capable of finding G-optimal designs for univariate settings so we need not repeat the exercise here. We used a swarm size of 150, following the lead of [32]. To account for computational burden, just 20 runs were performed. Though this is far fewer than the 500-runs of Nelder-Mead we implemented in Chapter 3, 20-runs of PSO captures far more information than the same number of runs with local optimizers. For greedy algorithms like PEXCH, the number of candidate optima is exactly equal to the number of runs, and entrapment in local optima is not only probable but

expected. PSO on the other hand, begins by considering 150 unique candidate designs. As the algorithm approaches convergence, each particle – which represents a candidate design in this case – tends toward the best design in the swarm. So while the combination of individual and shared information correlates the path of each particle, PSO is able to consider a much larger region of the search space when compared to the other algorithms in this thesis. We applied PSO to the $K = 4$, $N = 15$ design setting in addition to the cannonical cases of [1], but it took over a week for the script to complete and the G-efficiency of the proposed optimal design was no better than current SOA. Even so, just 20 runs for $K = 2$ provided excellent designs.

Figure 6.1 gives side-by-side boxplots for all $K = 2$ design cases, comparing the method of PSO and `Gloptipoly` to the method of Nelder-Mead and `Gloptipoly`. For these plots we compute relative efficiency with respect to the best-known designs of [32]; however, to fully illustrate the capabilities of this method we re-score these designs using `Gloptipoly`. As mentioned in Chapter 2, there is a small amount of error induced by the grid approximation, so by re-scoring the designs we ensure the greatest possible degree of accuracy.

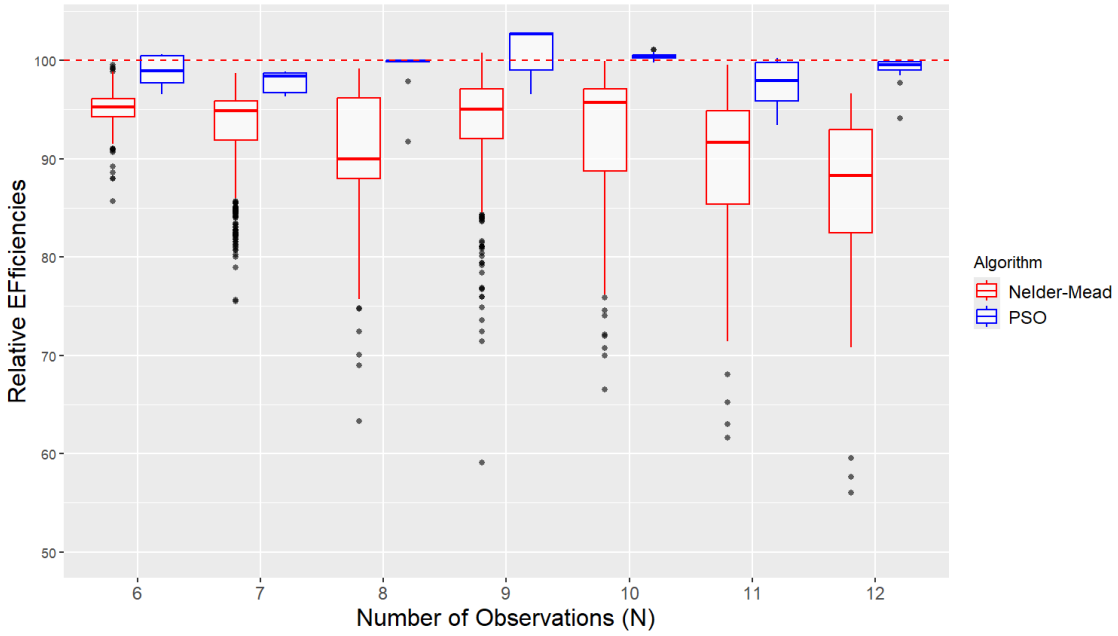

Figure 6.1: Boxplots for the distribution of designs by design scenario and algorithm, with efficiencies relative to the best designs of [32] for all two factor settings. The best design of just 20-runs of PSO surpassed the best design of 500 runs of Nelder-Mead in nearly every case.

For five out of the seven $K = 2$ cases, `Gloptipoly` and PSO were able to jointly find the best-known designs on the G-efficiency scale. For the other two cases, the designs we found were on-par

with SOA. Just 20 runs of the outer optimizer resulted in designs that beat the best of 500 runs of Nelder-Mead, suggesting that when we use intelligent outer optimizers like PSO that are robust to entrapment in local optima, we can effectively search the space of candidate designs with far fewer runs than [4] suggest – at least for these lower-dimensional cases.

Figure 6.2 visualizes the distribution of designs on the $K = 3$ cases, once again juxtaposing the two algorithms to provide a visual indication of relative performance. For $K = 3$, PSO only found the best-known design for $N = 10$ and $N = 13$, though the designs were highly efficient across the board. This highlights the unfortunate reality that as the dimension of the search space increases, the probability of entrapment in local optima increases, which necessitates additional runs of the outer optimizer to find the best-possible design. But even with our 18-core i9-10980XE CPU, $K = 3$, $N = 15$ took nearly 20 hours to complete 20-runs of PSO, so increasing the number of runs would need to be accompanied by an increase in compute power.

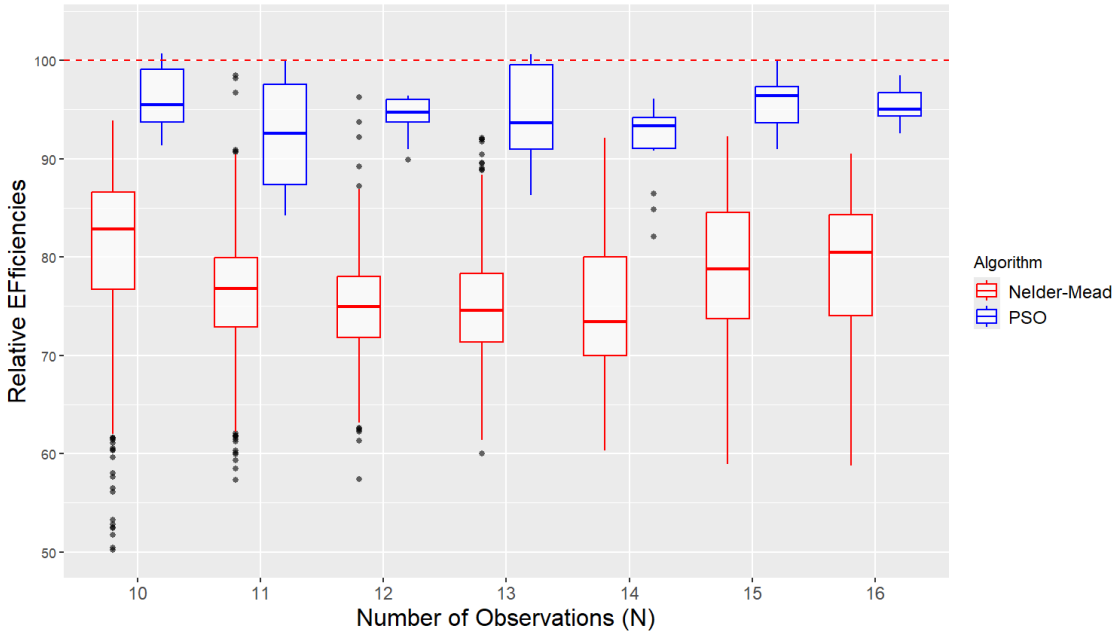

Figure 6.2: Boxplots for the distribution of designs by design scenario and algorithm, with efficiencies relative to the best designs of [32] for all three factor settings. The relative superiority of PSO becomes increasingly clear when assessing algorithmic efficacy on three-factor design scenarios.

6.4

Design Comparison by Algorithm

A question that naturally arises when comparing competing algorithms is how the proposed optimal designs compare to one another visually. This section considers all four methods that have been explored in this thesis: the Nelder-Mead Simplex Algorithm, PEXCH, the modified PEXCH, and PSO. Figure 6.3 visualizes SPV surfaces and design-points for the $K = 2$, $N = 8$ design setting. The designs proposed by all four algorithms place design-points in the corners, suggesting that we minimize worst-case scenario prediction variance by testing all four combinations of two factors at their extremes. The remaining four points are variable from design to design. PEXCH algorithms and Nelder-Mead all propose allocating the remaining design points asymmetrically; however, PSO, the clearly superior optimizer, has proposed a sampling methodology that seems to be closer to horizontal and vertical symmetry than the other designs. This seems to suggest that for the $K = 2$, $N = 8$ design setting, symmetric designs are highly desirable. Thus, future searches for G-optimal designs may consider this property a restriction on the space of candidate designs, dramatically reducing the size of the search space. Nonetheless, the geometry of the SPV surfaces and corresponding design-points is remarkably similar from design to design.

Since the point-exchange algorithms failed to find designs on-par with Nelder-Mead and PSO, they were not applied to design settings beyond those mentioned in Chapter 5. Next, we compare the designs proposed by Nelder-Mead to the designs proposed by PSO, assessing structural similarities in the placement of design-points. Figure 6.4 juxtaposes the Nelder-Mead proposed design (left) with the PSO proposed design (right) for $K = 2$, $N = 9, 10, 11$ cases. Structural similarities exist, but PSO continues to propose design points that seem to be less random, almost as if the Nelder-Mead proposed design is a not-yet-converged version of the PSO proposed design.

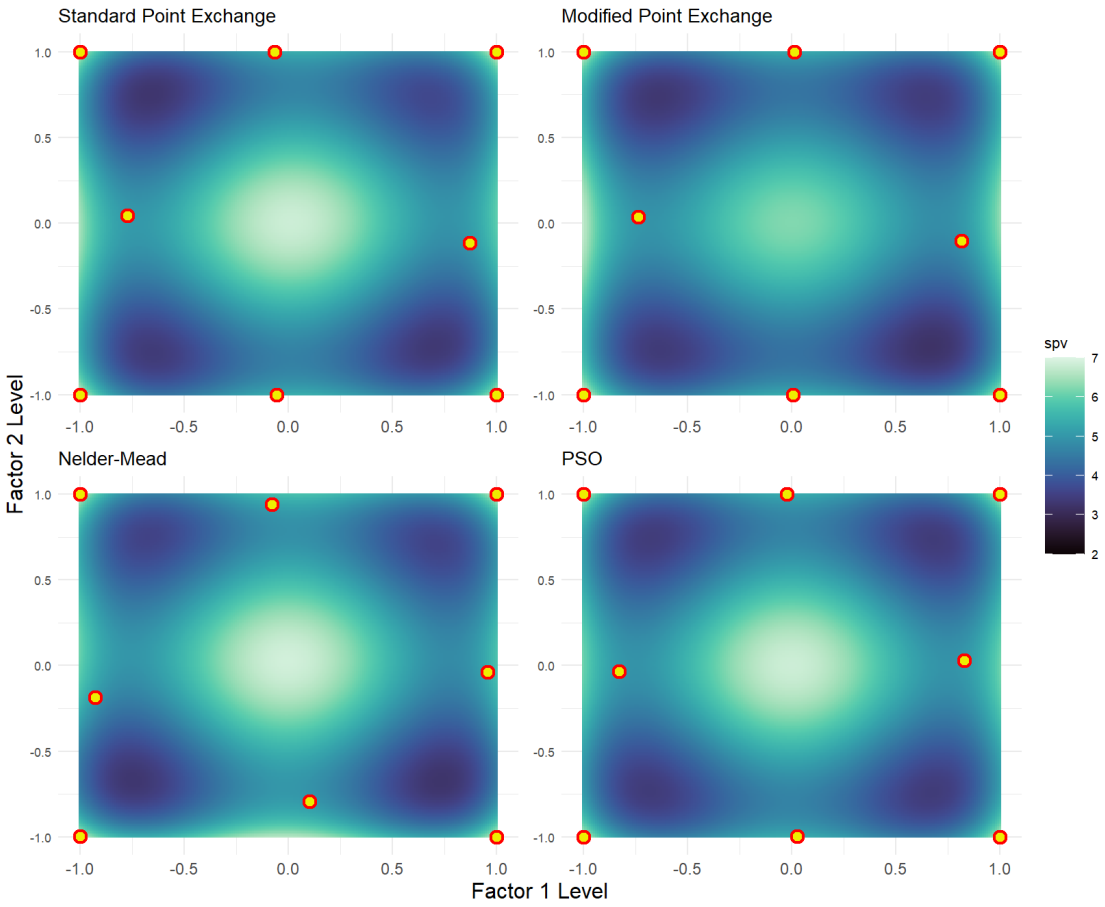

Figure 6.3: SPV-surfaces with superimposed design points for standard PEXCH (top left), modified PEXCH (top right), Nelder-Mead (bottom left), and PSO (bottom right). Geometrically, these designs share structure in both the curvature of SPV-surfaces and the placement of design-points.

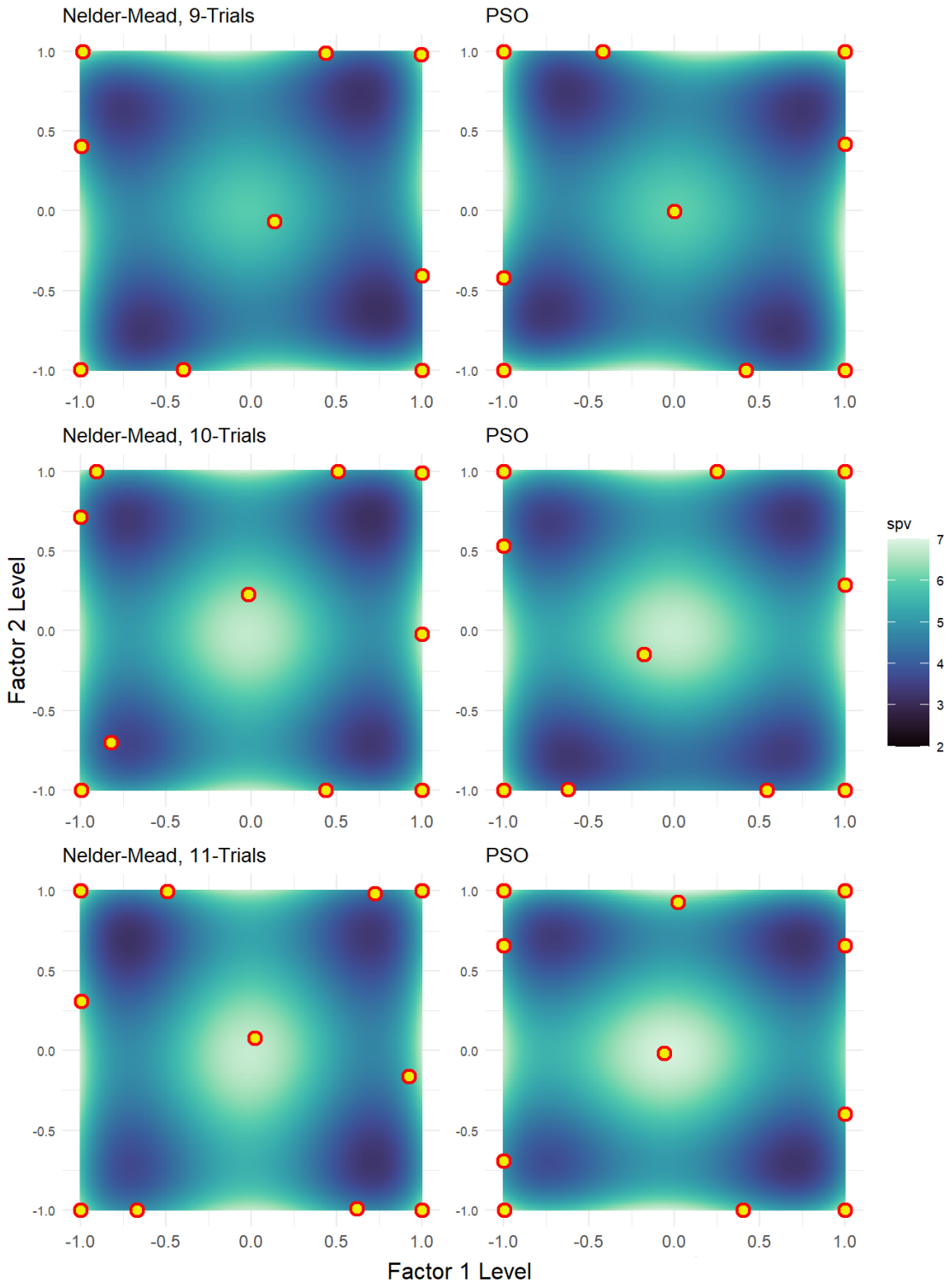

Figure 6.4: *SPV*-surfaces with superimposed design points for a selection of design-scenarios studied in this section. It is common for both Nelder-Mead and PSO to produce designs that sample the boundaries.

6.5

Conclusion and Future Work

In this thesis, we began with an overview of the G-optimal design problem, surveying the literature from the problem's inception in 1918 [25]– for which G-optimal designs were drived by hand in the univariate setting – to the SOA application of PSO [32]. After introducing notation and terminology, we introduced `Gloptipoly`, a MATLAB routine that formulates a hierarchy of semidefinite programs for which accumulation point of the set of optimal solutions is guaranteed to be the global optimum for the original problem, assuming the input function is a semidefinite polynomial. We provided proof-of-concept by re-scoring the designs of [32], identifying a small amount of approximation error in the proposed designs. A dense-grid approximation confirmed that `Gloptipoly` is providing more accurate max SPV values than the 5^K grid-approximation. We then embedded `Gloptipoly` in a custom-built objective-function evaulation tool, using Nelder-Mead as the outer optimizer. The resultant designs were on par with [1], but not quite on par with SOA for most design settings.

After reproducing designs for the cannoncial cases, we extended the research into optimal designs by proposing the first-ever G-optimal designs for response-surface models with higer-order interaction terms, cubic, and quartic models. These designs will need to be reproduced and improved upon by future scholars. Keeping with the traditions of the field, we implemented two computationally efficient point-exchange algorithms and demonstrated that antiquated optimizers are not suitable for this problem – good designs require more intelligent algorithms. Finally, we applied the SOA method, PSO, to the problem with `Gloptipoly` running under the hood to evaluate the objective function. While PSO + `Gloptipoly` make for a computationally burdensome marriage, we still managed to find the best-known designs for several design-scenarios, replicating and even improving upon the designs of [32]. The PSO proposed designs were compared to the Nelder-Mead proposed designs, and some structural similarities were identified.

Researchers interested in the G-optimal design problem should consider implementing the method of semidefinite relaxations proposed by [] in a more computationally efficient language, like Julia. Pairing PSO with `Gloptipoly` in a more efficient setting would make finding G-optimal designs for higher-order response-surface models much easier. Future researchers may also consider comparing the structural similarities of G-optimal designs proposed by various algorithms for higher order models. Lastly, mirroring the work of [22], researchers may choose to find sub-optimal designs with highly efficient algorithms like point-exchange, and use those designs as starting points

for more robust algorithms like Nelder-Mead or PSO. Alternatively, a practitioner may use the grid-approximation to compute max SPV until designs are nearly optimal, then switch to the computationally burdensome `Gloptipoly` to fine-tune towards the end of the design-search process.

REFERENCES

[1] John Borkowski. Using a genetic algorithm to generate small exact response surface designs. *Journal of Probability and Statistical Science*, 1, 01 2003.

[2] Johan Efberg. Yalmip : a toolbox for modeling and optimization in matlab. *2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No.04CH37508)*, pages 284–289, 2004. doi: https://doi.org/10.1109/CACSD.2004.1393890.

[3] Lenka Filová and Radoslav Harman. Ascent with quadratic assistance for the construction of exact experimental designs. *Computational Statistics*, 35(2):775–801, 2020. doi: https://doi.org/10.1007/s00180-020-00961-9.

[4] Peter Goos and Bradley Jones. *Optimal design of Experiments: A Case Study Approach*. Wiley, 2011. doi: https://doi.org/10.1002/9781119974017.

[5] Radoslav Harman and Lenka Filová. Computing efficient exact designs of experiments using integer quadratic programming. *Computational Statistics Data Analysis*, 71:1159–1167, 2014. doi: https://doi.org/10.1016/j.csda.2013.02.021.

[6] Didier Henrion and Jean Bernard Lasserre. Gloptipoly: global optimization over polynomials with matlab and sedumi. *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, 1:747–752 vol.1, 2002. doi: http://dx.doi.org/10.1145/779359.779363.

[7] Didier Henrion, Jean Bernard Lasserre, and Johan Lofberg. Gloptipoly 3: moments, optimization and semidefinite programming, 2007. doi: https://doi.org/10.48550/arXiv.0709.2559.

[8] Lucia N. Hernandez and Christopher J. Nachtsheim. Fast computation of exact g-optimal designs via i-optimality. *Technometrics*, 60(3):297–305, jul 2018. doi: https://doi.org/10.1080/00401706.2017.1371080.

[9] Onésimo Hernández-Lerma and Jean B. Lasserre. Approximation schemes for infinite linear programs. *SIAM Journal on Optimization*, 8(4):973–988, November 1998. doi: https://doi.org/10.1137/S1052623497315768.

[10] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995. doi: https://doi.org/10.1109/ICNN.1995.488968.

[11] J. Kiefer. Optimum experimental designs. *Journal of the Royal Statistical Society. Series B (Methodological)*, 21(2):272–319, 1959. doi: https://www.jstor.org/stable/2983802.

[12] J. Kiefer. Optimum designs in regression problems, ii. *The Annals of Mathematical Statistics*, 32(3):298–325, 1961. doi: https://www.jstor.org/stable/2237627.

[13] J. Kiefer and J. Wolfowitz. The equivalence of two extremum problems. *Canadian Journal of Mathematics*, 12(4):363–366, 1960. doi: https://doi.org/10.4153/CJM-1960-030-4.

[14] Henry Landau, editor. *Moments in Mathematics*. American Mathematical Society, 1987. doi: https://doi.org/10.1090/psapm/037.

[15] Jean B. Lasserre. Global optimization with polynomials and the problem of moments. *SIAM Journal on Optimization*, 11(3):796–817, January 2001. doi: https://doi.org/10.1137/S105262340036680.

[16] Jean B. Lasserre. Semidefinite programming vs. LP relaxations for polynomial programming. *Mathematics of Operations Research*, 27(2):347–360, May 2002. doi: https://doi.org/10.1287/moor.27.2.347.322.

[17] Jean B. Lasserre. A semidefinite programming approach to the generalized problem of moments. *Mathematical Programming*, 112(1):65–92, February 2007. doi: https://doi.org/10.1007/s10107-006-0085-1.

[18] Ruth K. Meyer and Christopher J. Nachtsheim. The coordinate-exchange algorithm for constructing exact optimal experimental designs. *Technometrics*, 37(1):60–69, 1995. doi: https://doi.org/10.2307/1269153.

[19] Ruth K. Meyer and Christopher J. Nachtsheim. The coordinate-exchange algorithm for constructing exact optimal experimental designs. *Technometrics*, 37(5):60–69, 1995. doi: https://doi.org/10.2307/1269153.

[20] Alan J. Miller and Nam-Ky Nguyen. Algorithm as 295: A fedorov exchange algorithm for d-optimal design. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 43(4):669–677, 1994. doi: https://doi.org/10.2307/2986264.

[21] Lenka Filová Radoslav Harman and Peter Richtárik. A randomized exchange algorithm for computing optimal approximate designs of experiments. *Journal of the American Statistical Association*, 115(529):348–361, 2020. doi: https://doi.org/10.1080/01621459.2018.1546588.

[22] Myrta Rodríguez, Bradley Jones, Connie M. Borror, and Douglas C. Montgomery. Generating and assessing exact g-optimal designs. *Journal of Quality Technology*, 42(1):3–20, jan 2010. doi: https://doi.org/10.1080/00224065.2010.11917803.

[23] Moein Saleh and Rong Pan. A clustering-based coordinate exchange algorithm for generating g-optimal experimental designs. *Journal of Statistical Computation and Simulation*, 86(8):1582–1604, August 2015. doi: https://doi.org/10.1080/00949655.2015.1077252.

[24] Alvaro Sanchez. Defining higher-order interactions in synthetic ecology: Lessons from physics and quantitative genetics. *Cell Systems*, 9(6):519–520, 2019. doi: https://doi.org/10.1016/j.cels.2019.11.009.

[25] Kirstine Smith. On the standard deviations of adjusted and interpolated values of an observed polynomial function and its constants and the guidance they give towards a proper choice of the distribution of observations. *Biometrika*, 12(1/2):1–85, 1918. doi: http://www.jstor.org/stable/2331929.

[26] Lu Lu Stephen J. Walsh and Christine M. Anderson-Cook. I-optimal or g-optimal: Do we have to choose? *Quality Engineering*, 36(2):227–248, 2024. doi: https://doi.org/10.1080/08982112.2023.2194963.

[27] Jos F Sturm. Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization methods and software*, 11(1-4):625–653, 1999. doi: https://doi.org/10.1080/10556789908805766.

[28] Stephen Walsh. Overview of optimal experimental design and a survey of its expanse in application to agricultural studies. 09 2022. doi: https://doi.org/10.26077/fc50-a735.

[29] Stephen J. Walsh. *Development and Applications of Particle Swarm Optimization for Constructing Optimal Experimental Designs*. PhD dissertation, Montana State University, Bozeman, MT 59715, 2021.

[30] Stephen J. Walsh, Thomsen B. Bolten, and John J. Borkowski. Generating optimal designs with user-specified pure replication structure. *Quality and Reliability Engineering International.*, 2024. doi: https://doi.org/10.1002/qre.3506.

[31] Stephen J. Walsh and John J. Borkowksi. Generating exact optimal designs via particle swarm optimization: Assessing efficacy and efficiency via case study. *Quality Engineering*, 2022. doi: https://doi.org/10.1080/08982112.2022.2127364.

[32] Stephen J. Walsh and John J. Borkowski. Fast computation of highly g-optimal exact designs via particle swarm optimization, 2022. doi: https://doi.org/10.48550/arXiv.2206.06498.

APPENDICES

APPENDIX

Supplementary Materials

CODE: A github repository containing all code used to produce the results and figures presented in this thesis can be found at the following link. Data generation was performed primarily in MATLAB and figures were produced using R.

`https://github.com/HyrumHansen/thesisResearch`

Chapter 3 Designs: The G-optimal designs we propose in chapter 3 may be found at the following link.

`https://github.com/HyrumHansen/thesisResearch/tree/main/borkowski_cases/designs`

Chapter 4 Designs: The G-optimal designs we propose in chapter 4 for models with higher-order interaction terms may be found at the following link,

`https://github.com/HyrumHansen/thesisResearch/tree/main/extension_functions/higher_order_interaction_data`

while the designs involving cubic and quartic RSM models may be found here.

`https://github.com/HyrumHansen/thesisResearch/tree/main/extension_functions/higher_order_data`

Chapter 5 Designs: The G-optimal designs we propose in chapter 5 using point-exchange algorithms may be found here,

`https://github.com/HyrumHansen/thesisResearch/tree/main/point_exchange/data`

Chapter 6 Designs: The G-optimal designs we propose in chapter 6 using PSO may be found here,

`https://github.com/HyrumHansen/thesisResearch/tree/main/pso_data`