5-2023

Multiparametric Continuous and Mixed-Integer Nonlinear Optimization with Parameters in General Locations

Andrew Pangia
apangia@clemson.edu

Multiparametric Continuous and Mixed-Integer Nonlinear Optimization with Parameters in General Locations

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematics

---

by
Andrew C. Pangia
May 2023

Accepted by:
Dr. Margaret M. Wiecek, Committee Chair
Dr. Yuyuan Ouyang
Dr. Matthew Saltzman
Dr. Boshi Yang

Abstract

Convex programming has been a research topic for a long time, both theoretically and algorithmically. Frequently, these programs lack complete data or contain rapidly shifting data. In response, we consider solving parametric programs, which allow for fast evaluation of the optimal solutions once the data is known. It has been established that, when the objective and constraint functions are convex in both variables and parameters, the optimal solutions can be estimated via linear interpolation.

Many applications of parametric optimization violate the necessary convexity assumption. However, the linear interpolation is still useful; as such, we extend this interpolation to more general parametric programs in which the objective and constraint functions are biconvex. The resulting algorithm can be applied to scalarized multiobjective problems, which are inherently parametric, or be used in a gradient dual ascent method. We also provide two termination conditions and perform a numerical study on synthetic parametric biconvex optimization problems to compare their effectiveness.

Dedication

As we have been plagiarising profusely from God's work throughout all these years, and the administration really does not care for religious references, we dedicate this humble compilation to Him, thank Him for all His help, and hope that our humble offering furthers His Kingdom.

'Failure is not an option – it is mandatory. The option is whether or not to let failure be the last thing you do.' -Maxim 70 of the Seventy Maxims of Maximally Effective Mercenaries (Howard Tayler, c. 2016)

Acknowledgments

Were we to list everyone without whom this thesis would be impossible, the acknowledgements would extend longer than the introduction. That being said, we must acknowledge our advisor for our Master's project, Dr. Gretchen Matthews, who hates acknowledgements. Thank you, Dr. Matthews; without you, none of this could ever have happened. Our PhD advisor, Dr. Margaret Wiecek, had to put up with our absurdities for a far longer time than the graduate school of Clemson intended, and our heartfelt thanks go out for that. Also worth noting are the efforts of Dr. Leo Rebholz, who loaned us a book on adaptive methods and error bounds thereof when we could find nothing conclusive in the literature. Thank you to you all. Also requiring mention is the United States Office of Naval Research, with Funding Grant number N00014-16-1-2725.

If you keep reading by this point, then you deserve what you get. In addition, we acknowledge the Potato; you are a moron, and we love you for it. We also cannot go without mentioning the von Loafenstein gang. Imma von Loafenstein, your neck must hurt from shaking your head so much. Dr. von Loafenstein, now you know where you are: your son is a lunkhead; you can go back to your paper. The retirement fund salesman ... need we say more? And lastly, Maurice von Loafenstein: ‘ “They'll never expect this” means ”I want to try something stupid.”’

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mathematical Programming, an important field of mathematics since the early twentieth century, has been thrust ever further into the limelight over the past fifteen years, with the advent of machine learning algorithms, and the mainstream infusion of statistical analysis methods (acts of minimization every one of them) into every walk of life. With such an influx of possible applications of mathematical programming, the subfields of multiobjective programming, and programming under uncertainty have burgeoned as well.

These fields necessarily require the encountering of costs and returns that are unknown for some period of time. These unknowns are called parameters, as opposed to variables, as they are not being solved for in the programs (like the variables), but they are present at all times. One response to these parameters is to wait for them to become known, and then solve the program. However, optimization is slow, especially for larger problems (consider the size of the program resulting from trying to minimize costs of running the train line through the Southeast Portion of the United States), so other options are being considered in a separate subfield of mathematical programming, called parametric programming. The goal of this field is to obtain closed-form solutions to the parametric programs as functions of the parameters (exponential, polynomial, logarithmic, etcetara as opposed to an optimization), so that, when the parameters are known, or as they change, the computation is far faster in order to redirect materials.

Parametric optimization has direct applications in pooling problems in chemical engineering, in which certain vats must maintain certain mixtures of chemicals, but the coefficients keep changing; also relevant to this field is the supply chain problem, and the power generation problem, different

situations which both use programs in which costs and demands are unknown, but must be met as they change, while costs must be kept to a minimum.

Another, more theoretical, application of parametric programming, is multiobjective programming, in which multiple functions with mutually exclusive optima are considered. One of the main methods for evaluating multiobjective programs (called MOPs) is to apply one of a myriad scalarization methods, in which parameters are introduced to weigh or constrain the different objectives.

We begin with an overview of the definitions necessary in Section 1.1, along with an evaluation of the state-of-the-art for parametric programming in Section 1.2

1.1 Definitions

sec:litDef

Prior to discussing the theory and algorithms found in the papers we have read, we begin by giving the following definitions, which will also be used when discussing the programs under consideration. Throughout this dissertation, Θ represents the space from which the parameters are drawn. Let $m, n, \kappa \in \mathbb{N}$, $\kappa \geq 2$, $\Theta \subseteq \mathbb{R}^{\kappa-1}$ be a bounded (convex) polyhedron, $\boldsymbol{\theta} \in \Theta$ represent the parameters, and $\boldsymbol{x} \in \mathbb{R}^n$ represent the decision variables. Note that, in practice, Θ will be partitioned into simplices with κ vertices. Let $\mathcal{P}(S)$ denote the power set of a set S, $\Theta \subseteq \mathbb{R}^{\kappa}$, and $\mathcal{X}(\Theta) \subseteq \mathbb{R}^n$.

Definition 1. *The* ***constraint set mapping*** *is*

$$\mathcal{X} : \Theta \mapsto \mathcal{X}(\Theta), \text{ defined by } \mathcal{X} : \boldsymbol{\theta} \to \mathcal{X}(\boldsymbol{\theta}),$$

where $\mathcal{X}(\boldsymbol{\theta})$ is a (feasible) set generated based on $\boldsymbol{\theta}$.

Definition 2. *The* ***feasible parameter set*** *is*

$$\Theta_f = \{\boldsymbol{\theta} \in \Theta : \mathcal{X}(\boldsymbol{\theta}) \neq \emptyset\}.$$

Definition 3. *The* ***extreme value function*** *is*

$$f^* : \Theta \mapsto f^*(\Theta) \text{ defined by } f^* : \boldsymbol{\theta} \to f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})} f(x, \boldsymbol{\theta}),$$

where $f : \mathbb{R}^n \times \Theta \to \mathbb{R}$.

Definition 4. *The* ***optimal set mapping*** *is*

$$\mathcal{X}^* : \Theta \mapsto \mathcal{X}^*(\Theta), \text{ defined by } \mathcal{X}^* : \boldsymbol{\theta} \to \mathcal{X}^*(\boldsymbol{\theta}) := \{x \in \mathcal{X}(\boldsymbol{\theta}) : f(x, \boldsymbol{\theta}) = f^*(\boldsymbol{\theta})\}.$$

Definition 5. *The* ***solubility set*** *is*

$$U = \{\boldsymbol{\theta} \in \Theta : \mathcal{X}^*(\boldsymbol{\theta}) \neq \emptyset\}.$$

Definition 6. *The* ***local stability set*** *is*

$$\mathcal{CR}(I, d) = \{\boldsymbol{\theta} \in U : ch(\mathcal{X}^*(\boldsymbol{\theta})) = I, dim(\mathcal{X}^*(\boldsymbol{\theta})) = d\},$$

where $ch(\mathcal{X}^*(\boldsymbol{\theta}))$ *is the index set denoting the constraints which are active at optimal solution set* $\mathcal{X}^*(\boldsymbol{\theta})$*, and* $I \subset \{1, \ldots, nCons\}$.

Note that, in the literature, the local stability set is referred to as a critical region or invariancy region. The following definitions for point-to-set mappings have been used in [12], [42], and [43], but have been taken from [43], as the text was clearest in that paper.

Definition 7. *A point-to-set map* $\mathcal{X} : \Theta \to \mathcal{P}(\mathbb{R}^n)$ *is considered to be* ***upper semi-continuous*** *at a given* $\hat{\boldsymbol{\theta}} \in \Theta$ *if for all open sets* $S \supseteq \mathcal{X}(\hat{\boldsymbol{\theta}})$*, there exists a neighbourhood* $N(\hat{\boldsymbol{\theta}})$ *such that* S *also contains* $\mathcal{X}(\boldsymbol{\theta})$ *for all* $\boldsymbol{\theta} \in N(\hat{\boldsymbol{\theta}})$ *(i.e.,* $\mathcal{X}(\boldsymbol{\theta}) \subseteq S$*).*

Definition 8. *A point-to-set map* $\mathcal{X} : \Theta \to \mathcal{P}(\mathbb{R}^n)$ *is considered to be* ***lower semi-continuous*** *at a given* $\hat{\boldsymbol{\theta}} \in \Theta$ *if for all open sets* S *sharing an element with* $\mathcal{X}(\hat{\boldsymbol{\theta}})$ *(*$S \cap \mathcal{X}(\boldsymbol{\theta}) \neq \emptyset$*), there exists a neighbourhood* $N(\hat{\boldsymbol{\theta}})$ *such that, for all* $\boldsymbol{\theta} \in N(\hat{\boldsymbol{\theta}})$*,* S *also shares an element with* $\mathcal{X}(\boldsymbol{\theta})$ *(i.e.,* $\mathcal{X}(\boldsymbol{\theta}) \cap S \neq \emptyset$*).*

Definition 9. *A point-to-set map* $\mathcal{X} : \Theta \to \mathcal{P}(\mathbb{R}^n)$ *is considered to be* ***continuous*** *at a given* $\hat{\boldsymbol{\theta}} \in \Theta$ *if* $\mathcal{X}$ *is both lower and upper semi-continuous at* $\hat{\boldsymbol{\theta}}$.

Definition 10. *A point-to-set map* $\mathcal{X} : \Theta \to \mathcal{P}(\mathbb{R}^n)$ *is considered to be a* ***convex map*** *on* Θ *if for all* $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \Theta$ *and for all* $\lambda \in (0, 1)$*,*

$$\lambda \mathcal{X}(\boldsymbol{\theta}_1) + (1 - \lambda)\mathcal{X}(\boldsymbol{\theta}_2) \subseteq \mathcal{X}(\lambda \boldsymbol{\theta}_1 + (1 - \lambda)\boldsymbol{\theta}_2).$$

*If we also require $\boldsymbol{\theta}_1 \neq \boldsymbol{\theta}_2$, then $\mathcal{X}$ is declared **essentially convex**.*

Note that the feasible regions in the nonlinear programs we discuss are convex point-to-set maps.

The following convexity terms are slightly different, but are both applicable to our research, so we state them here.

Definition 11. *Given convex spaces $\mathcal{X}(\Theta)$, Θ, and $\mathbb{R}$, a function $f : \mathcal{X}(\Theta) \times \Theta \to \mathbb{R}$ is **jointly convex** if f is convex considering the vector $(\boldsymbol{x}, \boldsymbol{\theta}) \in \mathcal{X}(\Theta) \times \Theta$.*

Definition 12. *Given convex spaces $\mathcal{X}(\Theta)$, Θ, and $\mathbb{R}$, a function $f : \mathcal{X}(\Theta) \times \Theta \to \mathbb{R}$ is **bi-convex** if f is convex with respect to $\boldsymbol{x}$ when holding $\boldsymbol{\theta}$ constant and if f is convex with respect to $\boldsymbol{\theta}$ when holding $\boldsymbol{x}$ constant.*

1.2 State of the Art

sec:litRev

We provide a brief summary of the literature foundational to our work. We begin with the theory, which came about in the latter half of the twentieth century, and then elaborate on the algorithmic advances in the twenty-first century.

1.2.1 Theory

The notations used in the following reviewed articles all vary slightly, so we attempt to make it consistent with that used in this paper.

Berge 1963 [18] The author wrote about topological spaces, and was checked only for the original statement and proof of the Maximum Theorem, stated below, which is restated for parametric programming specifically by Bank et al. [12] and Fiacco [42].

Theorem 1. *[18, p.115-116]*

1. *If $f : X \times \Theta \to \mathbb{R}$ is a lower semi-continuous function, and $\mathcal{X} : \Theta \to X$ is a lower semi-continuous mapping such that $\mathcal{X}(\boldsymbol{\theta}) \neq \emptyset$ for all $\boldsymbol{\theta} \in \Theta$, then the optimal value function*

$$f^*(\boldsymbol{\theta}) = \sup\{f(\boldsymbol{x}, \boldsymbol{\theta}) : \boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})\}$$

is lower semi-continuous.

2. *If $f : X \times \Theta \to \mathbb{R}$ is an upper semi-continuous function, and $\mathcal{X} : \Theta \to X$ is an upper semi-continuous mapping such that $\mathcal{X}(\boldsymbol{\theta}) \neq \emptyset$ for all $\boldsymbol{\theta} \in \Theta$, then the optimal value function*

$$f^*(\boldsymbol{\theta}) = \max\{f(\boldsymbol{x}, \boldsymbol{\theta}) : \boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})\}$$

is upper semi-continuous.

Mangasarian and Rosen 1964 [66] The authors examine convex programs which have constraints with random variables (modelled as parameters) on the right-hand side. Cited by [42], the authors prove Theorem 2 about jointly convex objective functions (Lemma 2 in [66]).

mangasarianThm

Theorem 2. *[66, p. 146] The optimal value function*

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x}} \; & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ \textit{s.t. } & \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0, \\ & \boldsymbol{\theta} \in \Theta \end{aligned}$$

is convex and continuous in $\boldsymbol{\theta}$ provided that f and $\boldsymbol{g}$ are both jointly convex and jointly continuous in $\boldsymbol{x}$ and $\boldsymbol{\theta}$.

It is worth noting that this theorem goes slightly further than the theorems presented by other authors by proving continuity of f^*, but the proof of continuity utilizes the convexity of f^*.

Evans and Gould 1970 [38] The authors provide a list of results detailing when optimal objective functions of optimization programs have continuity for the following problem. Let $\boldsymbol{x} \in \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^\kappa$, $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^\kappa$;

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \max_{\boldsymbol{x}} \; & f(\boldsymbol{x}) \\ \text{s.t. } & \mathbf{g}(\boldsymbol{x}) \leq \boldsymbol{\theta} \end{aligned}$$

This paper, cited by [55], was examined in a search for properties of a quartic objective function, but nothing was deemed helpful. The paper does, however, serve to establish continuity of $f^*(\boldsymbol{\theta})$.

Hogan 1972 [55] This is a quick, one page, one result paper which supplements the results found in [38].

hoganThm **Theorem 3.** *[55] Let $f : \mathbb{R}^n \to R$ and $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^k$ for some $k \in \mathbb{N}$, and $\boldsymbol{\theta} \in \Theta \subset \mathbb{R}^\kappa$; define*

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in X} & f(\boldsymbol{x}) \\ & s.t.\ \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{\theta}. \end{aligned}$$

If X is a compact convex set in $\mathbb{R}^n$, f and g_i, $i \in \{1, \ldots, k\}$ are both continuous on $X \times \mathbb{R}^n$, and g_i, $i \in \{1, \ldots, k\}$, are strictly convex on X for $\boldsymbol{\theta}$, then $f^*(\boldsymbol{\theta})$ is continuous on its domain.

As an example for why strict convexity is necessary, suppose $\mathbf{g}$ has an affine component (and is thus convex but not strictly convex). Let $f : \mathbb{R}^2 \to \mathbb{R}$, $\mathbf{g} : \mathbb{R}^2 \to \mathbb{R}^2$, where $f(x_1, x_2) = x_2$, $g_1(x_1, x_2) = x_1$, $g_2(x_1, x_2) = ||(x_1, x_2 - 1)||_2 + ||(x_1, x_2 + 1)||_2 - 2$, and $X = \{(x_1, x_2) : |x_1| + |x_2| \leq 2\}$. Take the parameter space to be $\Theta := \{(\theta_1, \theta_2) : \theta_1 \leq 0, \theta_2 = 2\sqrt{1 + {\theta_1}^2 - 2}\}$.

Note that X is a diamond centered at the origin. Then the program to be solved is

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{\boldsymbol{x} \in X} & x_2 \\ \text{s.t. } & x_1 \leq \theta_1, \\ & g_2(x_1, x_2) \leq \theta_2. \end{aligned}$$

This problem can be solved by taking the constraints to both be active and algebraically solving for x_2; observe that $\mathbf{g}(x_1, x_2)$ is largest when $x_1 = \theta_1$. Taking $\theta_1 = 0 = \theta_2$ yields that $x_2 = -1$, so $f^*(0, 0) = -1$, but, if $\theta_1 < 0$, then $f^*(\boldsymbol{\theta}) = 0$. Hence f^* is not continuous.

Bank et. al. 1982 [12] This book was examined very briefly to provide some definitions of terms. Let $\Theta \in \mathbb{R}^\kappa$ be a polyhedral parameter space. Let $\mathbf{g} : \mathbb{R}^n \times \mathbb{R}^\kappa \to \mathbb{R}^{nCons}$ be a vector-valued function, and consider the problem

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})} & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ \mathcal{X}(\boldsymbol{\theta}) := & \{\boldsymbol{x} : \mathbf{g}(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0\} \\ & \boldsymbol{\theta} \in \Theta. \end{aligned}$$

The following theorem on continuity is proven by [12].

bankCty **Theorem 4** (corollary 4.3.6.1 in [12]). *Let $I \subset \{1, \ldots, nCons\}$, and $d \in \{1, \ldots, n\}$ such that $\mathcal{CR}(I, d) \neq \emptyset$. Suppose, in the problem above, $f : \mathcal{X} \times \Theta \to \mathbb{R}$ and $\boldsymbol{g} : \mathcal{X} \times \Theta \to \mathbb{R}^{nCons}$ are jointly-convex and differentiable for all $i \in I$. Also suppose that $\mathcal{X}_f$ is lower semi-continuous on a proper subset, S, of critical region $\mathcal{CR}(I, d)$, then the following is true.*

1. *f^* is continuous on S;*
2. *$\mathcal{X}^*$ is closed on S;*
3. *There exists a vector-valued function $\boldsymbol{x} : S \to \mathbb{R}^n$ with $\boldsymbol{x}(\boldsymbol{\theta}) \in \mathcal{X}^*(\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in S$ which is continuous on S.*

Note that if $\mathcal{CR}(I, d)$ is not an open set, then the interior is a proper subset, so we can take $S = \text{int}(\mathcal{CR}(I, d))$.

Fiacco 1983 [42] The author accumulates all information known at the time about the following types of parametric programs. The first type has parameters only on the right-hand side of the constraints, while the second type has parameters everywhere. Constraints on the parameter space are not mentioned, so consider $\boldsymbol{\theta} \in \mathbb{R}^{\kappa}$, and $\boldsymbol{\theta}^1$, $\boldsymbol{\theta}^2$ to have dimension equal to the number of constraints.

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\text{s.t. } \mathbf{g}(\boldsymbol{x}) \leq \boldsymbol{\theta}^1, \\ &\quad \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{\theta}^2; \end{aligned}$$

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{\theta}) \\ &\text{s.t. } \mathbf{g}(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0, \\ &\quad \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{\theta}) = 0. \end{aligned}$$

Having stated some foundational theory and definitions, Fiacco restates the maximum theorem proven by Berge [18].

Theorem 5. *[42, p.14-15] If the constraint mapping $\mathcal{X} : \Theta \to \mathcal{X}(\Theta)$ is continuous such that $\mathcal{X}(\boldsymbol{\theta}) \neq \emptyset$ for all $\boldsymbol{\theta} \in \Theta$, and f is a continuous real-valued function defined on the space $\mathcal{X}(\Theta) \times \Theta$, then $f^* : \Theta \to \mathbb{R}$ defined by*

$$f^*(\boldsymbol{\theta}) = \inf_{\boldsymbol{x}} \{f(\boldsymbol{x}, \boldsymbol{\theta}) : \boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})\}$$

is continuous in Θ. In addition, the optimal set mapping $\mathcal{X}^*$ is an upper semi-continuous mapping.

Fiacco then proceeds to apply Lagrangian duality theory to obtain Taylor approximations for the decision variables within unspecified neighbourhoods of $\boldsymbol{\theta} = \mathbf{0}$. This can easily be extended (though the author refrains from doing so) to neighbourhoods of any given parameter value $\boldsymbol{\theta} = \boldsymbol{\theta}^0$. These neighbourhoods are the predecessors of the critical regions mentioned in the algorithms.

Fiacco and Ishizuka 1990 [43] The authors summarize results pertaining to the continuity and convexity of optimal value functions and optimal solution functions of general multiparametric nonlinear programs. Of interest to us is the following program.

fiacco90prob

$$f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{\theta}) \tag{1.1}$$
$$\text{s.t. } \boldsymbol{x} \in X(\boldsymbol{\theta}) := \{\boldsymbol{x} \in \mathbb{R}^n : \mathbf{g}(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0, \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{\theta}) = 0\}.$$

The authors cite [12], but portray the definitions of the terms in a clearer format. The definitions are listed in the beginning of this section, so will not be repeated here. The authors then state the following theorems from [44] regarding the convexity of the optimal value functions.

fiaccoIshiCvx

Theorem 6. *Suppose that $f : \mathcal{X}(\Theta) \times \Theta \to \mathbb{R}$ is jointly convex on $\mathcal{X}(\Theta) \times \Theta$, and that $\mathcal{X}$ is essentially convex on Θ. Then $f^* : \Theta \to \mathbb{R}$, defined by*

$$f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})} f(\boldsymbol{x}, \boldsymbol{\theta}),$$

is convex on Θ.

Theorem 7. *For (1.1), suppose that $\boldsymbol{g} : \mathcal{X} \to \mathbb{R}^{nCon}$ is a jointly quasiconvex function on $\mathbb{R}^n \times \Theta$ and $\boldsymbol{h}$ is jointly affine on $\mathbb{R}^n \times \Theta$. Then the constraint set mapping $\mathcal{X}$ is convex on Θ.*

In practice, though, joint (quasi)convexity frequently fails to hold, so the locations of parameters is greatly constrained when constructing programs adhering to this theorem.

Recall, for (1.1), that the first order necessary KKT conditions for a vector $(\boldsymbol{x}(\boldsymbol{\theta}), \boldsymbol{u}(\boldsymbol{\theta}), \boldsymbol{w}(\boldsymbol{\theta}))$ are as follows.

Definition 13. *Let the linear independence constraint qualifications (LICQ) hold for* (1.1)*, that is,* $\nabla_{\boldsymbol{x}} g_i(\boldsymbol{x}, \boldsymbol{\theta})$ *are linearly independent for all* $i \in I(\boldsymbol{x}, \boldsymbol{\theta}) := \{i \in \{1, \ldots, numConIneq\} : g_i(\boldsymbol{x}, \boldsymbol{\theta}) = 0\}$*. The First Order Necessary KKT Conditions are said to hold for* (1.1) *if for all local optimal solutions to* (1.1) $\boldsymbol{x}(\boldsymbol{\theta})$*, there exist* $\boldsymbol{u}(\boldsymbol{\theta})$ *and* $\boldsymbol{w}(\boldsymbol{\theta})$ *such that*

$$\begin{aligned} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}; \boldsymbol{\theta}) + \boldsymbol{u}^T \nabla_{\boldsymbol{x}} g(\boldsymbol{x}, \boldsymbol{\theta}) + \boldsymbol{w}^T h(\boldsymbol{x}, \boldsymbol{\theta}) &= \mathbf{0} \\ u_i g_i(\boldsymbol{x}, \boldsymbol{\theta}) &= \mathbf{0}, \\ \boldsymbol{u} &\geq \mathbf{0}. \end{aligned}$$

Theorem 8 (Theorem 5.1, p224 [43])**.** *For* (1.1)*, let* $\hat{\boldsymbol{\theta}} \in \Theta$*, and suppose that* x^* *is feasible, with associated Lagrange multipliers* $(\boldsymbol{u}^*, \boldsymbol{w}^*)$ *satisfying the KKT conditions stated above. Further assume that the second order sufficient conditions (SOSC) hold at* $(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{w}^*)$*, that is, assume*

$$\boldsymbol{z}^T \nabla^2_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{w}^*, \hat{\boldsymbol{\theta}}) \boldsymbol{z} > 0$$

for all $\boldsymbol{z} \neq 0$ *such that*

$$\begin{aligned} &\nabla_{\boldsymbol{x}} g_i(\boldsymbol{x}^*, \hat{\boldsymbol{\theta}}) \boldsymbol{z} \geq \mathbf{0} \text{ for constraints } i \in I(\boldsymbol{x}^*, \hat{\boldsymbol{\theta}}) \\ &\nabla_{\boldsymbol{x}} g_i(\boldsymbol{x}^*, \hat{\boldsymbol{\theta}}) \boldsymbol{z} = \mathbf{0} \text{ for } i \text{ with } u_i^* > 0 \\ &\nabla_{\boldsymbol{x}} h(\boldsymbol{x}^*, \hat{\boldsymbol{\theta}}) \boldsymbol{z} = \mathbf{0}. \end{aligned}$$

Suppose, even further, that the LICQ hold and the active constraints are nondegenerate, that is, $i \in I(\boldsymbol{x}^*, \hat{\boldsymbol{\theta}})$*. Then*

1. $\boldsymbol{x}^*$ *is an isolated local minimum of the program in* (1.1) *and the associated Lagrange multipliers* $(\boldsymbol{u}^*, \boldsymbol{w}^*)$ *are unique;*

2. *for* $\boldsymbol{\theta}$ *in a neighbourhood of* $\hat{\boldsymbol{\theta}}$*, there exists a unique once continuously differentiable vector function* $y(\boldsymbol{\theta}) := (\boldsymbol{x}(\boldsymbol{\theta}), \boldsymbol{u}(\boldsymbol{\theta}), \boldsymbol{w}(\boldsymbol{\theta}))^T$ *satisfying the KKT conditions and the SOSC at* $\boldsymbol{x}(\boldsymbol{\theta})$ *with* $[\boldsymbol{u}(\boldsymbol{\theta}), \boldsymbol{w}(\boldsymbol{\theta})]^T$ *such that* $\boldsymbol{y}(\hat{\boldsymbol{\theta}}) = (\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{w}^*)^T$ *and* $\boldsymbol{x}(\boldsymbol{\theta})$ *is a locally unique local minimum to*

the program in (1.1) *with associated unique Lagrange multipliers* $(\boldsymbol{u}(\boldsymbol{\theta}), \boldsymbol{w}(\boldsymbol{\theta}))$*;*

3. *The LICQ hold for* (1.1) *and* $u_i^* > 0$ *at* $\boldsymbol{x}(\boldsymbol{\theta})$ *in the aforementioned neighbourhood of* $\hat{\boldsymbol{\theta}}$*.*

Observe that, if (1.1) is strictly convex, then the second order sufficient conditions hold for all $\boldsymbol{z}$ as opposed to just the specified $\boldsymbol{z}$.

1.2.2 Algorithms

Again, let $m, n, \kappa, nEqCons, nCons \in \mathbb{N}$, $X \subseteq \mathbb{R}^n$, $\Theta \subseteq \mathbb{R}^\kappa$ be (convex) polyhedral. Further, let $f : \mathbb{R}^{m+n+\kappa} \to \mathbb{R}$, $\boldsymbol{h} : \mathbb{R}^{m+n+\kappa} \to \mathbb{R}^{nEqCons}$, $\mathbf{g} : \mathbb{R}^{m+n+\kappa} \to \mathbb{R}^{nCons}$. Letting $\boldsymbol{\theta}$ represent the parameters, and $\boldsymbol{x}$ and $\mathbf{y}$ represent the decision variables, consider the following optimization problem,

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x},\mathbf{y}} \; & f(\boldsymbol{x}, \mathbf{y}; \boldsymbol{\theta}) \\ \text{s.t. } & \boldsymbol{g}(\boldsymbol{x}, \mathbf{y}; \boldsymbol{\theta}) \leq 0 \\ & \boldsymbol{h}(\boldsymbol{x}, \mathbf{y}; \boldsymbol{\theta}) = 0 \\ & \boldsymbol{x} \in \mathcal{X}(\Theta), \mathbf{y} \in \{0,1\}^m, \boldsymbol{\theta} \in \Theta. \end{aligned} \tag{1.2}$$

genNLP

The literature presented in this section proposes solutions to specific cases of (1.2). The problems are grouped by type of objective function and location of parameters. We will maintain the notations used by the authors where necessary, but will standardize symbols where possible.

1.2.2.1 Multi-Parametric Mixed Binary Linear Programs

Pertsinidis 1992 [83] This work is examined due to its citation by future works, such as [1], [2], and [31]. The author discusses solving single parameter linear programs with the parameter in the right-hand side, that is $\theta \in [0, 1]$:

$$\begin{aligned} f^*(\theta) = \min_{\boldsymbol{x},\mathbf{y}} \; & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\mathbf{y} \\ \text{s.t. } & A\boldsymbol{x} + B\mathbf{y} \leq \boldsymbol{b} + r\theta; \\ & 0 \leq \theta \leq 1; \boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U; \mathbf{y} \in \{0,1\}^m. \end{aligned}$$

The author approaches the problem by first considering $\overline{\mathbf{y}}$ constant (by computing a single-point solution $(\overline{\boldsymbol{x}}, \overline{\mathbf{y}}, \overline{\theta})$ with, say, branch-and-bound) and using sensitivity analysis to form the corresponding piecewise-continuous solution $\overline{f}^*(\theta)$.

On forming $\overline{f}^*(\theta)$, the problem is then checked to see if any θ remain which give better solutions; on computing such θ, the process is repeated. This method is later used for the generation of more critical regions.

A linear approximation of nonlinear programs is also considered, using first-order Taylor approximations:

$$\begin{aligned} f^*(\theta) = \min_{\boldsymbol{x},\mathbf{y}} \; & f(\boldsymbol{x}) + \boldsymbol{d}^T\mathbf{y} \\ \text{s.t. } & A\boldsymbol{x} + \mathbf{g}(\mathbf{y}) \leq \boldsymbol{b} + r\theta; \\ 0 \leq \theta \leq 1; \; & \boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U; \mathbf{y} \in \{0,1\}^m. \end{aligned}$$

Acevedo 1997 [2] The authors improve the algorithm in [1] to apply to multiple parameters, $\kappa > 1$. Using parametric optimization techniques previously discovered (Gal 1972), they consider the problem

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{\boldsymbol{x},\mathbf{y}} \; & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\mathbf{y} \\ \text{s.t. } & A\boldsymbol{x} + B\mathbf{y} \leq b + E\boldsymbol{\theta} \\ & \boldsymbol{x} \in X, \, \mathbf{y} \in \{0,1\}^m, \, \boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^\kappa. \end{aligned} \tag{1.3}$$

mpLP

To solve the problem, they apply the following branch-and-bound (BB) algorithm.

1. Relax (1.3) to have $\mathbf{y} \in [0,1]^m$.
2. Select a variable $y_i \notin \{0,1\}$ on which to branch.
3. Re-solve (1.3) with the additional constraint $y_i = 0$ (then $y_i = 1$).
4. Compare each optimal value function with the previous upper bound optimal value function on each critical region (CR), replacing the upper bound if necessary.
5. Remove infeasible CRs and split the CRs each time the optimal value function changes.
6. Continue until all $\mathbf{y} \in \{0,1\}^m$ have been tested using BB techniques.

Acevedo 1998 [3] The authors have a parameter vector only in the right-hand side of the constraints, and not in the objective function.

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{\boldsymbol{x},\mathbf{y}} \; & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\mathbf{y} \\ \text{s.t. } & A\boldsymbol{x} + E\mathbf{y} \leq \boldsymbol{b} + F\boldsymbol{\theta} \\ & \boldsymbol{x} \in \mathbb{R}^n, \mathbf{y} \in \{0,1\}^\ell, \boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^\kappa. \end{aligned}$$

The paper [3] is a more rigorous restatement of [2] and references [76], who work with the above program for $\kappa = 1$.

Papalexandri 1998 [78] Papalexandri and Dimkou motivate their use of a multiparametric algorithm by representing vector-valued functions in multi-objective optimization using parameters to bound the other objective functions. Bounds on the parameters are found by solving the single-objective programs. The authors optimize mp-MILPs in the same vein as [2], but take advantage of the fact that the objective function is separable (contains no $x_i y_j$ terms) in order to apply a generalization of Benders' decomposition.

On computing a solution for a fixed integer vector $\bar{y}$, the authors create a master problem by taking fixed parameter points $\bar{\boldsymbol{\theta}} \in \Theta$ and use them to compute Lagrange multipliers to get lower bounds on the optimum function. The number of parameter points and the values taken are discussed in [78].

Dua 2001 [32] The authors optimize a linear program with the parameters in a hyper-rectangle. Take $\boldsymbol{\theta}_{\min}, \boldsymbol{\theta}_{\max} \in \mathbb{R}^\kappa$ to form $\Theta := \{\boldsymbol{\theta} \in \mathbb{R}^\kappa : \boldsymbol{\theta}_{\min} \leq \boldsymbol{\theta} \leq \boldsymbol{\theta}_{\max}\}$.

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x},\mathbf{y}} \; & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\mathbf{y} \\ \text{s.t. } & A\boldsymbol{x} + B\mathbf{y} \leq \boldsymbol{b} + E\boldsymbol{\theta} \\ & \boldsymbol{x} \in \mathbb{R}^n, \mathbf{y} \in \{0,1\}^\ell, \boldsymbol{\theta} \in \Theta := \{\boldsymbol{\theta} \in \mathbb{R}^\kappa : \boldsymbol{\theta}_{\min} \leq \boldsymbol{\theta} \leq \boldsymbol{\theta}_{\max}\}. \end{aligned}$$

They take the following approach:

1. Pick a fixed integer $\overline{\mathbf{y}}$ and solve the resulting parametric LP to obtain a piecewise linear objective function $\overline{f}^*(\boldsymbol{\theta})$ over a set of critical regions CR_i, $i \in \{1, 2, \ldots, numCRs\}$.

2. On each subrectangle CR_i, treat $\boldsymbol{\theta}$ as a variable and solve the integer linear program (using B&B or similar methods) excluding previous $\overline{\mathbf{y}}$ to obtain solution $(x_i^*, y_i^*, \boldsymbol{\theta}_i^*)$, and forcing the objective to be better than $\overline{f}(\boldsymbol{\theta})$. This is done with the following added constraint:

$$\boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\mathbf{y} \leq \overline{f}(\boldsymbol{\theta});$$

a more cumbersome constraint excluding previous $\overline{\mathbf{y}}$ can be found in [32].

3. Return to Step 1 using $\overline{\mathbf{y}} := \mathbf{y}^*$ to obtain more critical regions.

Li and Ierapetritou 2007 [64] The authors consider a program of the following form for $\boldsymbol{\theta} \in \mathbb{R}$, similar to [70], but lacking inequality constraints:

$$\begin{aligned} f^*(\theta) = \min_{\boldsymbol{x},\mathbf{y}} \; & \boldsymbol{c}^x(\theta)^T\boldsymbol{x} + \boldsymbol{c}^y(\theta)^T\mathbf{y} \\ \text{s.t. } & A^x(\theta)\boldsymbol{x} + A^y(\theta)\mathbf{y} = \boldsymbol{b}(\theta), \\ \boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U, & \end{aligned}$$

where $\boldsymbol{x} \in \mathbb{R}^n$, $\mathbf{y} \in \{0,1\}^m$, $\boldsymbol{c}^x : \mathbb{R} \to \mathbb{R}^n$, $\boldsymbol{c}^y(\theta) : \mathbb{R} \to \mathbb{R}^m$, and A^{ix}, A^{iy} are properly sized matrix functions of θ. They apply a BB algorithm which begins by applying a nonparametric BB at a specific value of $\boldsymbol{\theta}$ to select an initial upper bound.

Mitsos and Barton 2009 [70] The authors consider the following program for $\theta \in \mathbb{R}$,

$$\begin{aligned} f^*(\theta) = \min_{\boldsymbol{x},\mathbf{y}} \; & \boldsymbol{c}^x(\theta)^T\boldsymbol{x} + \boldsymbol{c}^y(\theta)^T\mathbf{y} \\ \text{s.t. } & A^{1x}(\theta)\boldsymbol{x} + A^{1y}(\theta)\mathbf{y} \leq \boldsymbol{b}^1(\theta), \\ & A^{2x}(\theta)\boldsymbol{x} + A^{2y}(\theta)\mathbf{y} \leq \boldsymbol{b}^2(\theta), \\ & \boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U, \boldsymbol{x} \in \mathbb{R}^n, \mathbf{y} \in \{0,1\}^m \end{aligned}$$

where $\boldsymbol{c}^x : \mathbb{R} \to \mathbb{R}^n$, $\boldsymbol{c}^y(\theta) : \mathbb{R} \to \mathbb{R}^m$, and A^{ix}, A^{iy} are properly sized matrix functions of θ. They apply a BB algorithm which begins by solving the relaxation program where $\mathbf{y} \in [0,1]^m$ to check feasibility of the system over θ, and then, at each node, find the parameter range for which that node's solution is optimal.

1.2.2.2 Multi-Parametric Mixed Binary Bilinear Programs

Faısca 2008 [39] The authors optimize a function nonlinear with respect to $\boldsymbol{x}$ and $\boldsymbol{\theta}$, and $\mathbf{y}$ and $\boldsymbol{\theta}$ but with parameters in the constraints on the right-hand side.

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{\boldsymbol{x},\mathbf{y}} \ & (\boldsymbol{c} + H\boldsymbol{\theta})^T \boldsymbol{x} + (\boldsymbol{d} + L\boldsymbol{\theta})^T \mathbf{y} \\ \text{s.t. } & A\boldsymbol{x} + E\mathbf{y} \leq \boldsymbol{b} + F\boldsymbol{\theta} \\ & \Gamma\boldsymbol{x} + \Phi\mathbf{y} = \gamma + \Psi\boldsymbol{\theta} \\ & \boldsymbol{x} \in \mathbb{R}^n, \mathbf{y} \in \{0,1\}^m, \boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^\kappa. \end{aligned}$$

Note Γ, Φ, and Ψ are constant matrices, and γ is a constant vector for linear equality constraints; Θ is a polyhedral set, that is, $\Theta := \{\boldsymbol{\theta} \in \mathbb{R}^\kappa : G\boldsymbol{\theta} \leq \boldsymbol{e}\}$, where G is some constant matrix and e some constant vector. Properties of the problem due to KKT conditions are explored at specific parameter values, using Corollary 3.2.3 from [42, p76-77]. Then the following algorithm is used.

1. Find a global minimum $(\boldsymbol{x}^*, \mathbf{y}^*, \boldsymbol{\theta}^*)$ using $\boldsymbol{\theta}$ as a free variable.
2. Fix $\overline{\mathbf{y}} = \mathbf{y}^*$ and use Corollary 3.2.3 from [42, p76-77] to reduce the problem to an mp-LP; how to do this is the focus of the paper. Solve this mp-LP to obtain a current piecewise continuous solution $\overline{f}(\boldsymbol{\theta})$.
3. Repeat Step 1 and find a new global minimum excluding anything worse than $\overline{f}(\boldsymbol{\theta})$ and excluding previously utilized values of $\mathbf{y}$.

Wittmann-Hohlbein 2013 [99] The authors discuss optimizing a function with nonconvex constraints which are bilinear in the parameter and the variables.

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{\boldsymbol{x},\mathbf{y}} \ & (\boldsymbol{c} + H\boldsymbol{\theta})^T \boldsymbol{x} + (\boldsymbol{d} + L\boldsymbol{\theta})^T \mathbf{y} \\ \text{s.t. } & A(\boldsymbol{\theta})\boldsymbol{x} + E(\boldsymbol{\theta})\mathbf{y} \leq \boldsymbol{b} + F\boldsymbol{\theta} \\ & \boldsymbol{x} \in \mathbb{R}^n, \mathbf{y} \in \{0,1\}^q, \boldsymbol{\theta} \in \Theta := \{\boldsymbol{\theta} \in \mathbb{R}^s : \boldsymbol{\theta}_{\min} \leq \boldsymbol{\theta} \leq \boldsymbol{\theta}_{\max}\}. \end{aligned}$$

The bilinear approximations discussed in the background information are used to force the parameters into the right-hand side of the constraints, where the program becomes the previously solved programs.

Oberdieck et. al. 2014 [75] The authors optimize a function as in [39], but with parameters only in the right-hand side. This is a BB version of [39].

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{x,y}\ & (c + H\boldsymbol{\theta})^T x + (d + L\boldsymbol{\theta})^T y \\ \text{s.t. } & Ax + Ey \leq b + F\boldsymbol{\theta} \\ & x \in \mathbb{R}^n, y \in \{0,1\}^q, \boldsymbol{\theta} \in \{\boldsymbol{\theta} \in \mathbb{R}^s : \boldsymbol{\theta}_{\min} \leq \boldsymbol{\theta} \leq \boldsymbol{\theta}_{\max}\}. \end{aligned}$$

Note that the nonlinear terms are all jointly bilinear in x and $\boldsymbol{\theta}$, so can be approximated, following the method from [67], known as McCormick relaxation. Further note is the concurrence of this with [99]; enough authors are similar, that most likely reviewers were off.

1.2.2.3 Multi-Parametric Non-Linear Programs

Acevedo 1996 [1] Acevedo and Pistikopoulos consider the case with $k = 1$ and apply an approximation algorithm to the following version of (1.2), where the parameter is a scalar value appearing linearly only in the inequality constraints, and the binary variables are linear in both the objective function and the constraints. The vector-valued functions f, g, and h are potentially nonlinear, but assumed to be convex and differentiable.

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{\boldsymbol{x},\mathbf{y}}\ & f(\boldsymbol{x}) + \boldsymbol{d}^T y \\ \text{s.t. } & \boldsymbol{h}(x) = 0 \\ & \boldsymbol{g}(x) + B\mathbf{y} \leq \boldsymbol{b} + r\boldsymbol{\theta} \\ & \boldsymbol{x} \in X,\ \mathbf{y} \in \{0,1\}^m,\ \boldsymbol{\theta} \in [\boldsymbol{\theta}_{\min}, \boldsymbol{\theta}_{\max}]. \end{aligned} \tag{1.4}$$

spNLP

The authors compute piece-wise linear approximations of the optimal function $f^*(\boldsymbol{x}, \mathbf{y}; \boldsymbol{\theta})$ as follows.

1. Select a specific binary vector $\bar{\mathbf{y}}$
2. Compute specific optimal solutions to (1.4) at $\bar{y}$ using the endpoints $\boldsymbol{\theta}_{\min}$ and $\boldsymbol{\theta}_{\max}$, and interpolate an overestimate due to convexity of the objective function.
3. Next, compute underestimates using two first-order Taylor approximations centered at $\boldsymbol{\theta}_{\min}$

and $\boldsymbol{\theta}_{\max}$. The required gradients are computed using the KKT first-order-necessary-conditions (FONC).

4. Check that the maximum distance between the lower and upper bounds does not exceed some desired tolerance ϵ.

5. Split $[\boldsymbol{\theta}_{\min}, \boldsymbol{\theta}_{\max}]$ at the interior point, $\boldsymbol{\theta}_{int}$, where the two lower bounds intersect, and then repeat steps (2) through (5) on the resulting subintervals.

6. Use the underestimates to create an MILP excluding $\bar{y}$ which can then be solved using BB to try to find a better binary vector to replace $\bar{y}$. If one is not found, end.

7. Return to step (2).

Dua et. al. 1999 [31] Here the authors discuss three different algorithms for solving mp-01MINLPs, citing [1], [2], and [78] for the Generalized Benders' Decomposition.

Pistikopoulos et. al. 2001 [84] The authors optimize a continuous convex quadratic parametric program with parameters only in the right-hand side of affine constraints. Let $Q \in \mathbb{R}^{n \times n}$ be positive semi-definite, and A and F be appropriately sized matrices.

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x}} \ & \frac{1}{2}\boldsymbol{x}^T Q \boldsymbol{x} \\ \text{s.t. } & A\boldsymbol{x} \leq \boldsymbol{b} + F\boldsymbol{\theta} \\ & \boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^{\kappa}. \end{aligned}$$

Pistikopoulos et. al. proceed as follows.

1. Treat $\boldsymbol{\theta}$ as a variable and find a global minimum solution $(x^*, \boldsymbol{\theta}^*)$.

2. Compute KKT points for the program based on $(x^*, \boldsymbol{\theta}^*)$, and compute the corresponding Hessian and gradient.

3. Use the Hessian and gradient to create a linear approximation, $\hat{f}(\boldsymbol{\theta})$, of the true optimal function for the program.

4. Return to Step 1 with Θ being reset to be the portion of Θ on which $\hat{f}(\boldsymbol{\theta})$ is not minimal.

Dua et. al. 2002 [29] Utilizing a linearization approach, the authors address the following problem.

$$\begin{aligned} \min_{\boldsymbol{x}} \ & \frac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + c^T\boldsymbol{x} \\ \text{s.t. } & A\boldsymbol{x} + \boldsymbol{b} + F\boldsymbol{\theta} \leq \mathbf{0} \\ & \boldsymbol{x} \in X, \boldsymbol{\theta} \in \Theta, \end{aligned}$$

Where $X \subset \mathbb{R}^n$ and Θ are polyhedral, and Q is symmetric positive definite.

Let $x^* : \Theta \to \mathbb{R}^n$ define an optimal solution function which creates the objective function $f^* : \Theta \to \mathbb{R}$, let $\Theta_f \subseteq \Theta$ be the domain of f^*, and let $b(\boldsymbol{\theta})$ be an affine function of $\boldsymbol{\theta}$.

quadCty **Theorem 9** (Quadratic NLP Continuity [29])**.** *Consider the mp-QP with Q positive definite and Θ convex.*

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in X} \ & \frac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + \boldsymbol{c}^T\boldsymbol{x} \\ & \textit{s.t. } A\boldsymbol{x} \leq b(\boldsymbol{\theta}) \\ \boldsymbol{x} \in X \subseteq \mathbb{R}^n & \\ \boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^\kappa. & \end{aligned}$$

Then the set of feasible parameters, Θ_f, is convex, the optimal solution function $x^*(\boldsymbol{\theta}) : \Theta \to \mathbb{R}^n$ is continuous and piecewise affine, and the optimal value function $f^*(\boldsymbol{\theta}) : \Theta_f \to \mathbb{R}$ is continuous, convex, and piecewise quadratic.

This theorem is a summary of properties for a specific problem based on theorems stated in [43], which have to do with parameters only in the right-hand side of constraints (and are thus not listed).

In addition, the authors give a reasonably understandable definition of critical regions and how they are computed. Specifically, the primal and dual solutions $\boldsymbol{x} : \Theta \to \mathbb{R}^n$ and $\boldsymbol{u} : \Theta \to \mathbb{R}^{\text{rank}(A)}$ can be solved for due to the structure of the problem (see the proof of Theorem 9 in [29] for details). Then certain constraints are inactive, and those constraints (along with the dual nonnegativity constraints) intersected with the original parameter space create the critical region corresponding to that solution. When the inactive constraints change, or the dual variables become negative, then

a new solution $(\boldsymbol{x}, \boldsymbol{u})$ is found for a new critical region.

Dua and Pistikopoulos 2002 [33] The authors aggregate properties of optimality functions for optimal functions and optimal decision functions for convex and non-convex multi-parametric binary mixed-integer nonlinear programs (mp-01MINLPs). They first formulate an algorithm to solve convex multi-parametric binary mixed-integer quadratic programs (mp-MIQPs) with affine constraints and parameters in the right-hand side of the constraints. This problem type has been enumerated multiple times in this paper, see the summary on [31]; indeed, this algorithm follows very closely to the algorithms in [31], so will not be repeated.

Using the approximation methods in [6] and [8] (examined in Section 1.2.3), the authors create an algorithm for solving nonconvex programs as follows (f and g are both nonconvex):

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{x,y}\ & f(x,y) \\ \text{s.t. } & \mathbf{g}(x,y) \leq \boldsymbol{b} + F\boldsymbol{\theta} \\ \boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U \in \mathbb{R}^n, & \mathbf{y} \in \{0,1\}^m, \boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^\kappa. \end{aligned}$$

This problem gets approximated using the approximation from the α-BB method found in [8], and is solved in the same way.

Bemporad & Filippi 2006 [14] Bemporad et. al. create a more general algorithm for solving convex programs; note that f and $\mathbf{g}$ are jointly convex in x and $\boldsymbol{\theta}$:

$$\begin{aligned} f^*(\boldsymbol{\theta}) = \min_{\boldsymbol{x}}\ & f(\boldsymbol{x};\boldsymbol{\theta}) \\ \text{s.t. } & \mathbf{g}_i(\boldsymbol{x};\boldsymbol{\theta}) \leq \mathbf{0}; \quad i \in \{1,\ldots,nCons\}; \\ & A\boldsymbol{x} = \boldsymbol{b} + F\boldsymbol{\theta}; \\ & \boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^\kappa. \end{aligned}$$

The algorithm developed here heavily inspired the Approximate Simplex Method (ASM) from [62]. This algorithm involves estimating the parameter space with simplices. As will be detailed later, the above program is solved over each simplex, denoted CR, as follows.

1. Let $\{\vartheta_0, \vartheta_1, \vartheta_2, \ldots, \vartheta_{\kappa+1}\}$ be the vertices of CR.

2. At each $j \in \{0, 1, 2, \ldots, \kappa + 1\}$, compute solutions, x_j, to

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}; \vartheta_j) \\ &\text{s.t. } g_i(\boldsymbol{x}; \vartheta_j) \leq 0; \quad i \in \{1, \ldots, nCons\}; \\ &A\boldsymbol{x} + B\vartheta_j + \boldsymbol{d} = 0. \end{aligned}$$

3. Generate $f^*(\boldsymbol{\theta})$ as a linear interpolation of $\boldsymbol{\theta}$ using X and $\boldsymbol{\vartheta}$.

4. If $f^*(\boldsymbol{\theta})$ is not accurate enough, split CR into more simplices. Continue to the next critical region until there are no more critical regions to check.

Domínguez et. al. 2010 [27] The authors aggregate four algorithms which solve mp-NLPs:

1. The outer-approximation algorithm from [31];

2. A quadratic algorithm similar to the one found in [84];

3. The Approximate Simplex Method from [14];

4. A geometric vertex search algorithm proposed by [73], which is too far removed from the objective of this research and is thus not reviewed here.

Domínguez 2013 [28] The authors extend the results from [31] to approximately solve the following general program.

$$\begin{aligned} &\min_{\boldsymbol{x},\mathbf{y}} \boldsymbol{c}^T\mathbf{y} + f(\boldsymbol{x}; \boldsymbol{\theta}) \\ &\text{s.t. } \mathbf{g}(\boldsymbol{x}) + E\mathbf{y} \leq \boldsymbol{b} + F\boldsymbol{\theta} \\ &\boldsymbol{\theta} \in \Theta \end{aligned}$$

The continuous portion of the objective function, $f(\boldsymbol{x})$, is quadratically approximated at a global minimum $(\boldsymbol{x}^*, \mathbf{y}^*, \boldsymbol{\theta}^*)$, where $\boldsymbol{\theta}$ is considered as extra variables. In addition, $g(\boldsymbol{x})$ is linearly approximated, which results in a polyhedral parameter set.

The algorithm closely mirrors [31] with variations on finding infeasibilities in Θ. Two variants are found: compute infeasibilities at each vertex of the critical region, which can get compu-

tationally expensive, given the potential number of vertices; alternatively, compute a convex hull using outer-approximations of the constraints, which is less accurate.

Axehill et. al. 2014 [11] Axehill and his fellow authors extend the BB algorithm found in [2] to apply to the multi-parametric mixed-integer quadratic problem (mp-MIQP)

$$\begin{aligned} \min_{\boldsymbol{x},\mathbf{y}} \ & \begin{bmatrix} \boldsymbol{x}^T & \mathbf{y}^T \end{bmatrix} Q \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} + \boldsymbol{c}^T \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} \\ \text{s.t. } & A \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} = \boldsymbol{b} + F\boldsymbol{\theta} \qquad (1.5) \quad \boxed{\texttt{mpNLP}} \\ & \boldsymbol{x} \in X, \mathbf{y} \in \{0,1\}^m, \boldsymbol{\theta} \in \Theta. \end{aligned}$$

In addition to said extension, the authors also utilize the concept of σ-suboptimality.

Definition 14. *A solution $\boldsymbol{z}^* = (\boldsymbol{x}^*, \boldsymbol{y}^*)$ is said to be σ-**suboptimal** to an optimization problem with a corresponding objective function value $\hat{z}$ that satisfies $\hat{\boldsymbol{z}} - \boldsymbol{z}^* \leq \sigma$ where $\sigma > 0$, $\boldsymbol{z}^* < \infty$.*

Utilizing σ-suboptimality decreases runtime of the algorithm, as the algorithm will be only searching for a "good enough" solution as opposed to the best solution. The main idea of the algorithm is as follows.

1. Relax (1.5) to have $\mathbf{y} \in [0,1]^m$.
2. Solve the relaxation to obtain upper-bound functions.
3. Select a variable $y_i \notin \{0,1\}$ on which to branch.
4. Re-solve (1.5) with the additional constraint $y_i = 0$ (then $y_i = 1$).
5. Compare each optimal value function with the constraint with the previous upper bound optimal value function on each critical region (CR). If a constrained optimal value function is within suboptimality of the the upper bound, keep it and make it the new upper bound.
6. Remove infeasible CRs and split the CRs each time the optimal value function changes.
7. Continue until all $\mathbf{y} \in \{0,1\}^m$ have been tested using BB techniques.

Adelgren 2016 [4] The author designs an algorithm for multi-parametric quadratic programs with parameters in general positions. Consider an objective function quadratic in x, affine in $\boldsymbol{\theta}$, with constraints linear in x and affine in $\boldsymbol{\theta}$. Consider the following program, where Θ is a polyhedral parameter set, and $Q(\boldsymbol{\theta})$ is a positive definite matrix affine in $\boldsymbol{\theta}$.

$$\begin{aligned} &\min \tfrac{1}{2}\boldsymbol{x}^T Q(\boldsymbol{\theta})\boldsymbol{x} + c(\boldsymbol{\theta})^T\boldsymbol{x} \\ &\text{s.t. } \boldsymbol{x} \in \{\boldsymbol{x} \in \mathbb{R}^n : A(\boldsymbol{\theta})\boldsymbol{x} - \boldsymbol{b}(\boldsymbol{\theta}) \leq \boldsymbol{0}, \boldsymbol{x} \geq \boldsymbol{0}\}, \boldsymbol{\theta} \in \Theta. \end{aligned}$$

Using slack variables, rewrite the constraint set to be equalities $\{A(\boldsymbol{\theta})\boldsymbol{x} - \boldsymbol{b}(\boldsymbol{\theta}) + \boldsymbol{s} = \boldsymbol{0}, \boldsymbol{x} \geq \boldsymbol{0}\}$ where $s \in \mathbb{R}^m_{\geq}$. Applying the KKT conditions yields the following system of equations, called the multi-parametric Linear Complementarity Problem (mpLCP), where $\boldsymbol{u}^1$ and $\boldsymbol{u}^2$ denote the dual variables associated with the constraints.

$$\begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{u}^2 \end{bmatrix} - \begin{bmatrix} \boldsymbol{0} & -A(\boldsymbol{\theta}) \\ A(\boldsymbol{\theta})^T & Q(\boldsymbol{\theta}) \end{bmatrix} \begin{bmatrix} \boldsymbol{u}^1 \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}(\boldsymbol{\theta}) \\ \boldsymbol{c}(\boldsymbol{\theta}) \end{bmatrix}$$

$$\begin{bmatrix} \boldsymbol{s} & \boldsymbol{u}^2 \end{bmatrix} \begin{bmatrix} \boldsymbol{u}^1 \\ \boldsymbol{x} \end{bmatrix} = \boldsymbol{0}$$

$$\boldsymbol{x} \geq \boldsymbol{0}, \boldsymbol{s} \geq \boldsymbol{0}, \boldsymbol{u}^1 \geq \boldsymbol{0}, \boldsymbol{u}^2 \geq \boldsymbol{0}.$$

The author also develops what is believed to be the first algorithm to solve mpLCPs. This work continues in [5].

Benyamin 2019 [17] The authors apply Groebner bases to the NLP

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathbb{R}^n_{\geqq}} & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ \text{s.t. } & g_i(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0,\ i \in \{1, \ldots, m\} \\ & h_j(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{0}, \end{aligned}$$

where f, g_i, and h_j must be polynomial in $\boldsymbol{\theta}$; the idea is that the KKT conditions can be solved using Groebner bases, which are an algebraic construct derived as bases for finite-dimensional polynomial rings (consider them bases as for a vector space).

Qiu et. al. 2019 [86] The authors work with model predictive control (MPC) optimization, and use polynomial interpolation, specifically the Galerkin method, to construct closed-form approximations to the solutions of the programs derived from the KKT conditions.

Charitopoulos 2020 [23] At the same time as [17], the authors of this thesis-turned-book apply Groebner bases to extend parametric programming to apply to the following program for a variety of engineering problems.

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathbb{R}^n_{\geqq}} \; & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ & \text{s.t. } g_i(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0, \; i \in \{1, \ldots, m\} \end{aligned}$$

with the only limit being that the functions f and g_i must be polynomial in $\boldsymbol{\theta}$ so that Groebner basis theory applies.

Pappas 2021 [80] The authors process quadratically constrained quadratic programs which contain terms bilinear in $\boldsymbol{x}$ and $\boldsymbol{\theta}$, but no higher; they do this by considering the KKT conditions and the different combinations of the active constraints.

Pistikopoulos 2021 book [85] The authors amalgamate their results from all their previous work with parametric quadratic programs and elaborate on the different options for solving integer programs; as this is a textbook, it provides an aggregation, but no new information.

Leverenz et. al. 2022 [63] The authors apply a subgradient descent algorithm developed for the general case in [7] to the following program, with f and $\mathbf{g}$ assumed to be jointly convex in $\boldsymbol{x} \in \mathbb{R}^n$ and $\theta \in \Theta \subseteq \mathbb{R}^\kappa$:

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathbb{R}^n_{\geqq}} \; & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ & \text{s.t. } g_i(\boldsymbol{x}, \boldsymbol{\theta}) \leq \mathbf{0}, \; i \in \{1, \ldots, m\} \\ & A\boldsymbol{x} + B\boldsymbol{\theta} + \boldsymbol{d} = \mathbf{0}. \end{aligned}$$

Zewde 2022 [100] The authors use a penalty function, a method not implemented since [42], to solve the general nonlinear program

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathbb{R}^n_{\geqq}} \; & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ \text{s.t. } & \mathbf{g}(\boldsymbol{x}, \boldsymbol{\theta}) \leq \mathbf{0}, \\ & \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{\theta}) = \mathbf{0}. \end{aligned}$$

They assume convex nonlinear constraints, but their examples contain $\boldsymbol{\theta}$ only in the right-hand side of the constraints.

1.2.3 Miscellaneous

misc

Androulakis et. al. 1995 [8] The authors investigate solving nonconvex problems with continous variables and without parameters of the following form:

$$\begin{aligned} \min_x \; & f(x) \\ s.t. \; & h_j(x) = 0, \quad j \in \{1, \ldots, nEqCons\} \\ & g_k(x) \leq 0, \quad k \in \{1, \ldots, nCons\} \\ & Ax \leq b \\ & x^L \leq x \leq x^U. \end{aligned}$$

In this algorithm, called the α-BB algorithm, each nonconvex term of f is replaced by a convex approximation. Some terms, such as bilinear terms, have special structures that allow for specific approximations, however, in general, the nonconvex terms $f_i(x)$ are replaced by a polynomial interpolation multiplied by a large enough constant α_i for each $i \in \{1, \ldots, numTerms\}$ which increases the Hessian of the objective function. Note α is chosen to be larger in absolute value than the minimum eigenvalue (which is negative) of the Hessian of the nonconvex term.

The branching, similar to integer BB, occurs on each component of x, bisecting the interval $[x_i^L, x_i^U]$ to some desired interval size.

Adjiman et. al. 2000 [6] The authors adapt [8] to work on mixed integer nonlinear programs:

$$\begin{aligned} &\min_{x,y} f(x,y) \\ s.t.\ &g_k(x,y) \leq 0, \quad k \in \{1,\ldots,nCons\} \\ &h_j(x,y) = 0, \quad j \in \{1,\ldots,nEqCons\} \\ &Ax + Ey \leq b \\ &x^L \leq x \leq x^U; y \in \{0,1\}^m. \end{aligned}$$

A modified BB algorithm is used, where branching occurs on components of y first, and branching occurs on components of x afterwards.

Dua et. al. 2004 [30] This paper is a case study of specific multiparametric nonconvex problems and how to estimate them with upper bounds and with lower bounds:

$$\begin{aligned} f^*(\theta) = \min_x &f(x) \\ s.t.\ &g_i(x) \leq F_i\theta, \qquad i \in \{1,\ldots,nCons\} \\ &x^L \leq x \leq x^U, \theta \in \Theta. \end{aligned}$$

Specifically, the bilinear case is examined, and, while general methods are mentioned, the authors direct interested readers to investigate [8] and [6].

Gorski et. al. 2007 [48] The authors focus on biconvexity of functions, with definitions and propositions. In addition, an algorithm for solving programs involving such functions in the objective (the constraint sets form X and Y and are thus convex) is given. This algorithm, called the Alternate Convex Search by the authors, alternates between optimizing for x given y, and, having solved for x, solving for y until convergence is reached. This algorithm is proven to converge by the authors. More valuably for our work, the authors also cite [45], who provide an algorithm for solving nonparametric biconvex programs.

1.3 Research Goals

By and large, the authors of the algorithms above consider convex constraints and objectives without the necessary distinction between joint and bi- convexity; for example, $x\theta$ is biconvex, but not jointly convex, in x and θ; [14] makes special note of joint convexity in their interpolation algorithm, while [48] discusses biconvex optimization without parameters. In order to consider uncertain costs or returns, one must make the distinction between joint convexity (convexity in the vulgate) and biconvexity. We begin our work with MOPs as they contain direct applications of parametric optimization. Indeed, when scalarized, the MOP becomes a parametric program whose optimal parametric solutions are efficient for the MOP. Currently, in multiobjective optimization literature, this benefit of parametric optimization has been recognized theoretically [37, 49], but only in the initial stages computationally, for MOLPs, MOQPs, MOMILPs, BOMIQPs [53, 57, 74, 88, 92, 54]. The supporting parametric algorithms only enabled the application to simple MOPs of the above forms, and, as such, we consider it important to present our theoretical and numerical expansion here. We apply the algorithm from [14], ASM, in multiobjective programming, in spite of its lack of guarantee to work with scalarizations such as the weighted sum (which adds parameters into the costs in the objective functions). Following on the unguaranteed success of ASM from [14], we extend it to apply to programs where the objective and constraints are biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$. With that extension in hand, we apply a BB algorithm to mixed integer programs of the same form.

1.4 Research Road Map

We originally intended to construct a BB algorithm to solve parametric programs by using subgradient ascent to relax complicating constraints into the objective function where they can be more easily worked with. However, as we proceeded, we realized that the algorithms which we had access to, namely the ASM and Quadratic Approximation, were not rated for the programs which would result from applying a primal-dual method with subgradient ascent. As a result, we took a detour to explore application of the ASM to MOPs, which led us to realise the overall robustness of the algorithm. Consequently, we extended its underlying theory with a view to use it as the subroutine in the primal-dual algorithm. Along the way, we discovered that the primal-dual algorithm itself has practical convergence issues— not to be confused with theoretical convergence, which is proven in [7]. Therefore, we directly utilize the ASM extension for the BB algorithm.

1.5 Research Contributions

We successfully demonstrate that approximation methods are worth considering in MOPs due to their ability to solve much faster than more analytic methods and with comparable accuracy. However, the current approximations that we found fail to function properly for functions biconvex in the variables and parameters. Given that the most popular scalarization method, the weighted-sum method, creates precisely such functions, we explore an extension of the approximations which can interact with such functions with mathematical rigour.

We conduct an extensive review of the literature for parametric programming, in which we find nothing related to error bounds. Given that our approximation is a linear interpolation of a function, indeed, a parametric optimization is a function of its parameters, we proceed to investigate the theoretical error bounds, both *a priori* and *a posteriori*, for interpolations, most frequently with respect to partial differential equations (PDE). We find that our particular problem can be modeled as a problem of variation, that is, an extension of PDEs with different boundary conditions; problems of variation have no upper bounds on errors. As a result, we construct an upper bound using the biconvex parallel to the upper bound for convex functions. Being an upper bound for a problem of variation, this is automatically a good upper bound. This upper bound grants us the ability to extend the ASM from [14] to apply to programs with constraints and objectives biconvex in the parameters and variables; we designate this extension the Biconvex Approximate Simplex Method (BASM).

Being aware of the widespread nature of parametrically biconvex functions, we further explore parametric NLPs with a discussion of a generalized quadratic approximation algorithm as well as the shortcomings of subgradient ascent algorithms for parametric programming. We follow up this discussion with an implementation of a BB algorithm with the BASM as the subroutine used to solve the continuous program at each node.

At each stage of our research, we work to strengthen the current paradigm of parametric algorithm research by including tests of each algorithm on randomly generated instances of parametric quadratically constrained quadratic programs, designed based on the examples found on [26]. By putting forth these tests, we hope to provide a foundational benchmark for others to build off of and campare their algorithms against.

1.6 Dissertation Layout

We open with Chapter 2, in which we compare the linear interpolation method designed in [14] against the multiparametric Linear Complementarity Program method designed in [4]. We demonstrate that, given the runtime comparisons, the approximation is worth considering further, which we do in Chapter 3. We then use the Biconvex Approximate Simplex Method in Chapter 4 to emphasize the potential advancements in mixed-integer programming.

Chapter 2

On Solving Parametric Multiobjective Quadratic Programs with Parameters in General Locations

ch:MOP

[This chapter consists of material which has been published in 2022 in the *Annals of Operations Research* as [58] with coauthors Drs. Pubudu Wijesiri Jayasekara and Margaret Wiecek. Except where used for comparison, the below material is the author's own contributions to the aforementioned paper.]

2.1 Introduction

Many real-life problems in engineering, business, and management are characterized by multiple conflicting criteria such as cost, performance, reliability, safety, productivity, affordability, and the field of multiobjective optimization provides models, theories and methods to address these types of applications [35]. In addition to conflict between objective functions, uncertainty—from unknown or imprecise data, inaccurate measurements, or inadequate models—is another important charac-

teristic of many real-life decision problems. In the operations research literature, there are three classical paradigms to model uncertainty: probabilistic, possibilistic, and deterministic. The latter approach, using crisp sets to define domains within which uncertainties vary, has given foundation to robust optimization and parametric optimization.

In addition to constants, which are taken from known data, and variables, which are unknown but under user's control and being solved for, a parametric optimization problem also contains parameters that represent unknown quantities being beyond the interested party's control and not being solved for. Parameters fundamentally change the problem: the parametric single objective program (mpSOP) is solved to obtain a solution vector-valued function and the corresponding optimal value function, both of which map the parameters to the solution space. In contrast, the nonparametric SOP is solved to obtain a specific solution vector and corresponding optimal value. Similarly, the parametric multiobjective program (mpMOP) is solved to obtain a parametrized collection of efficient (Pareto) sets, as opposed to a specific efficient (Pareto) set. Parametric multiobjective optimization offers a bridge to robust multiobjective optimization, which uses various concepts to yield robust efficient solutions arguably preferred under the conditions of uncertainty. Since robust efficient solutions correspond to specific values of uncertainty, the robust approach is subsumed in the parametric approach and the latter emerges as a more universal methodology [95].

Studies on mpMOPs go back to 1979 when Naccache [72] examined the stability of solution sets due to perturbations in the feasible set. Since then, researchers have worked on parametrization of feasible sets, objective functions and domination structures, and related stability properties [13, 36, 56, 65, 81, 82, 90]. Parametric linear programs are analyzed in [16, 19], while the polyhedral structure of the efficient set for such problems is more recently examined in [40, 93]. Theoretical works have been accompanied by applied studies. Methods to compute a family of solution sets for unconstrained problems with a scalar parameter are developed in [25, 61, 97] and applied to mechatronic systems in which the parameter plays the role of time [98]. The theoretical framework for a solution method based on a technique which subdivides the solution and parameter spaces is proposed in [89]. In engineering design, genetic algorithms are tailored to the parametric case to allow for design exploration in the presence of exogenous factors [46, 51].

Independently of parametric multiobjective optimization, parametrization of the efficient set of MOPs can be conducted as a result of treating scalarized MOPs as mpSOPs [96]. This point of view is theoretically examined in [37, 49] and used for computational work in [53, 54, 57, 88, 92]

to obtain different types of parametric descriptions of the efficient set for MOPs arising in portfolio optimization. In [74], such a description is obtained for multiobjective quadratic programs (MOQPs) whose objective functions are linearized.

While theoretical and computational studies on mpMOPs have been steadily progressing, the latter have been rather limited despite the fact that parametric optimization can provide a complete parametric description of the efficient set regardless of uncertainty in the model. Because mpMOPs can be solved before their efficient sets are actually needed, in time-sensitive situations, the only computations required are function evaluations at the specific parameter values stemming from the situation. This benefit, however, comes at the cost of the increased computational complexity which parametric optimization causes. This paper puts forward the premise that parametrization of the efficient set can naturally be combined with solving mpMOPs because the algorithms performing the former can also be used to achieve the latter. The purpose is to examine the state-of-the-art in algorithmic development for parametric multiobjective quadratic programs (mpMOQPs). Based on this premise, mpMOQPs are scalarized to be solved as parametric (single objective) quadratic programs by means of suitable algorithms designed for this class of problems. We develop a generalized weighted-sum scalarization method that subsumes several established scalarizations and leads to several related SOPs that can be matched with different solution algorithms. In a computational study, we compare the performance of three parametric optimization algorithms on mpQPs with linear and/or quadratic constraints that result from different scalarizations. The algorithms are also applied to mpMOQPs modeling decision-making problems in statistics and portfolio optimization. By means of these applications, the interplay between the scalarizations is disclosed and additional insight into the parametric efficient solution sets is obtained.

The paper is structured as follows. In Section 2.2, we formulate the mpMOQP and define solution concepts. The generalized weighted-sum method and related scalarizations for MOPs are developed in Section 2.3. In the subsequent two sections we present algorithms for solving different types of mpQPs that result from various scalarizations of mpMOQPs. In Section 2.4 we present an algorithm for mpQPs with quadratic and linear constraints. In each section, results from computational tests performed on synthetic problems are included. Section 2.5 contains the applications and the paper is concluded in Section 2.6. Supplementary information is included in the Appendix.

2.2 Problem Statement

sec:theory

We define the mpMOQP and the solution concepts used to solve this class of problems. We also introduce the assumptions that are needed by the solution algorithms we present in the subsequent sections.

Let $\kappa, n, r, \tilde{r} \in \mathbb{N}$, $\tilde{r} \leq r$, and $\mathbb{R}^\kappa, \mathbb{R}^n, \mathbb{R}^r$ be Euclidean spaces that are related to the parameter space, decision or solution space, and objective or outcome space, respectively. Let $\Theta \subseteq \mathbb{R}^\kappa$ be a parameter space, $\mathcal{X} : \mathbb{R}^\kappa \to \mathbb{R}^n$ be a point-to-set map such that $\mathcal{X}(\Theta) = \bigcup_{\boldsymbol{\theta} \in \Theta} \mathcal{X}(\boldsymbol{\theta}) \subseteq \mathbb{R}^n$, and $\mathcal{X}(\boldsymbol{\theta}) \neq \emptyset$ for all $\boldsymbol{\theta} \in \Theta$. We investigate the following mpMOQP:

$$\begin{aligned}
\min_{\boldsymbol{x}} \mathbf{f}(\boldsymbol{x};\boldsymbol{\theta}) = [f_1(\boldsymbol{x};\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{x}^T Q_1(\boldsymbol{\theta})\boldsymbol{x} + \mathbf{p}_1(\boldsymbol{\theta})^T\boldsymbol{x} + c_1(\boldsymbol{\theta}),\ \ldots,\ f_{\tilde{r}}(\boldsymbol{x};\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{x}^T Q_{\tilde{r}}(\boldsymbol{\theta})\boldsymbol{x} + \mathbf{p}_{\tilde{r}}(\boldsymbol{\theta})^T\boldsymbol{x} + c_{\tilde{r}}(\boldsymbol{\theta}),& \\
f_{\tilde{r}+1}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{p}_{\tilde{r}+1}(\boldsymbol{\theta})^T\boldsymbol{x} + c_{\tilde{r}+1}(\boldsymbol{\theta}),\ \ldots,\ f_r(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{p}_r(\boldsymbol{\theta})^T\boldsymbol{x} + c_r(\boldsymbol{\theta})] \ \ (\text{MOQP}(\boldsymbol{\theta})) & \quad \text{mpmoqp} \\
\text{s.t. } \boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta}) = \{\boldsymbol{x} \in \mathbb{R}^n : A(\boldsymbol{\theta})\boldsymbol{x} \leqq \boldsymbol{b}(\boldsymbol{\theta}), \boldsymbol{x} \geqq \mathbf{0}\} & \\
\boldsymbol{\theta} \in \Theta, &
\end{aligned}$$

where $A : \Theta \to \mathbb{R}^{m \times n}$, $\boldsymbol{b} : \Theta \to \mathbb{R}^m$. The vector valued-objective $\mathbf{f} : \mathbb{R}^n \times \Theta \to \mathbb{R}^r$ is composed of functions $f_i : \mathbb{R}^n \times \Theta \to \mathbb{R}$ such that $Q_i : \Theta \to \mathbb{R}^{n \times n}$, $\mathbf{p}_i : \Theta \to \mathbb{R}^n$ and $c_i : \Theta \to \mathbb{R}$ for all $i = 1, \ldots, r$. The vector of parameters $\boldsymbol{\theta}$ models quantities that are unknown due to lack of knowledge at the time of the MOP construction. Examples include road capacity, interest rate, selling price, air humidity, material density or other application-specific values. Throughout this paper we make the following assumptions.

ass:general **Assumption 1.** *1. The parameter space $\Theta \subseteq \mathbb{R}^\kappa$ is a nonempty compact and polyhedral set.*

2. The feasible set $\mathcal{X}(\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in \Theta$ is a nonempty compact and convex set.

To perform optimization, we need to be able to compare the outcomes of MOQP($\boldsymbol{\theta}$).

def:inequality **Definition 15.** *1. Let $\bar{\boldsymbol{\theta}} \in \Theta$ be fixed and $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{X}(\bar{\boldsymbol{\theta}})$. Then $\boldsymbol{f}(\boldsymbol{x}^1;\bar{\boldsymbol{\theta}})(<)(\leqq) \leq \boldsymbol{f}(\boldsymbol{x}^2;\bar{\boldsymbol{\theta}})$ if and only if $f_i(\boldsymbol{x}^1;\bar{\boldsymbol{\theta}})(<) \leq f_i(\boldsymbol{x}^2;\bar{\boldsymbol{\theta}})$ for all $i = 1, \ldots, r$, where $\leq$ requires strict inequality for at least one index i, while $\leqq$ allows equality for all i.*

2. Let $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{X}(\Theta)$ and $i \in \{1, \ldots, r\}$. Then $f_i(\boldsymbol{x}^1;\Theta) \leq f_i(\boldsymbol{x}^2;\Theta)$ if and only if $f_i(\boldsymbol{x}^1;\boldsymbol{\theta}) \leq f_i(\boldsymbol{x}^2;\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in \Theta$.

3. *Let* $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{X}(\Theta)$. *Then* $\boldsymbol{f}(\boldsymbol{x}^1;\Theta)(<)(\leqq) \leq \boldsymbol{f}(\boldsymbol{x}^2;\Theta)$ *if and only if* $\boldsymbol{f}(\boldsymbol{x}^1;\boldsymbol{\theta})(<)(\leqq) \leq \boldsymbol{f}(\boldsymbol{x}^2;\boldsymbol{\theta})$ *for all* $\boldsymbol{\theta} \in \Theta$.

Solving MOQP($\boldsymbol{\theta}$) for a fixed parameter $\boldsymbol{\theta} = \bar{\boldsymbol{\theta}} \in \Theta$ is defined as finding the set of (weakly) efficient solutions. To conserve space, below we provide the very similar definitions of weakly efficient solutions and sets concurrently with the definitions of efficient solutions and sets via the use of parentheses. We apply the same format to define (weak) Pareto outcomes and sets.

def:eff_one **Definition 16.** *Let* $\bar{\boldsymbol{\theta}} \in \Theta$ *be fixed. A feasible solution* $\hat{\boldsymbol{x}} \in \mathcal{X}(\bar{\boldsymbol{\theta}})$ *is called (weakly) efficient to MOQP(*$\bar{\boldsymbol{\theta}}$*) for if there exists no other solution* $\boldsymbol{x} \in \mathcal{X}(\bar{\boldsymbol{\theta}})$ *such that* $\boldsymbol{f}(\boldsymbol{x};\bar{\boldsymbol{\theta}})(<) \leq \boldsymbol{f}(\hat{\boldsymbol{x}};\bar{\boldsymbol{\theta}})$. *Let* $\mathcal{X}_{(w)E}(\bar{\boldsymbol{\theta}})$ *denote the set of (weakly) efficient solutions at* $\boldsymbol{\theta} = \bar{\boldsymbol{\theta}}$.

We assume that $\mathcal{X}_E(\bar{\boldsymbol{\theta}}) \neq \emptyset$ for each $\bar{\boldsymbol{\theta}} \in \Theta$. Solving MOQP($\boldsymbol{\theta}$) for all $\boldsymbol{\theta} \in \Theta$ is defined as finding the (weakly) efficient set $\mathcal{X}_E(\boldsymbol{\theta}) \subseteq \mathcal{X}(\boldsymbol{\theta})$ for each $\boldsymbol{\theta} \in \Theta$.

def:eff_all **Definition 17.** *The set* $\mathcal{X}_{(w)E} \subseteq \mathcal{X}(\Theta)$, *defined as the collection of the (weakly) efficient sets* $\mathcal{X}_{(w)E}(\boldsymbol{\theta})$, $\mathcal{X}_{(w)E} := \{X_{(w)E}(\boldsymbol{\theta})\}_{\boldsymbol{\theta}\in\Theta}$, *is called the set of (weakly) efficient solutions to MOQP(*$\boldsymbol{\theta}$*) for all* $\boldsymbol{\theta} \in \Theta$.

Let $\bar{\boldsymbol{\theta}} \in \Theta$ be fixed. We define the attainable set, $\mathcal{Y}(\bar{\boldsymbol{\theta}})$, as the image of the feasible set $\mathcal{X}(\bar{\boldsymbol{\theta}})$ under the vector-valued objective function mapping $\mathbf{f}$, that is,

$$\mathcal{Y}(\bar{\boldsymbol{\theta}}) := \{\mathbf{y} \in \mathbb{R}^r : \mathbf{y} = \mathbf{f}(\boldsymbol{x}, \bar{\boldsymbol{\theta}}), \boldsymbol{x} \in \mathcal{X}(\bar{\boldsymbol{\theta}})\}.$$

The image of a (weakly) efficient solution to MOQP($\bar{\boldsymbol{\theta}}$) for $\bar{\boldsymbol{\theta}} \in \Theta$ is called a (weak) Pareto outcome. Let $\mathcal{Y}_{(w)P}(\bar{\boldsymbol{\theta}})$ denote the set of all (weak) Pareto outcomes for $\bar{\boldsymbol{\theta}} \in \Theta$.

def:Pareto_all **Definition 18.** *The set* $\mathcal{Y}_{(w)P} \subseteq \mathbb{R}^r$, *defined as the collection of the (weak) Pareto sets* $\mathcal{Y}_{(w)P}(\boldsymbol{\theta})$, $\mathcal{Y}_{(w)P} := \{Y_{(w)P}(\boldsymbol{\theta})\}_{\boldsymbol{\theta}\in\Theta}$, *is called the set of (weak) Pareto outcomes to MOQP(*$\boldsymbol{\theta}$*) for all* $\boldsymbol{\theta} \in \Theta$.

Scalarization methods can reformulate MOQP($\boldsymbol{\theta}$) into mpSOPs using parameters that are specific to each method. In effect, the resulting mpSOPs have two types of parameters, those from the original model, $\boldsymbol{\theta}$, and the auxiliary parameters, $\boldsymbol{\epsilon}$ and $\boldsymbol{\lambda}$, needed for scalarization. We refer to the former as modeling parameters and to the latter as scalarization parameters. We further discuss the scalarization parameters in Section 2.3. The mpSOPs are solved with algorithms that provide two types of information: (i) a partition of the augmented parametric space into subsets

called invariancy regions (critical regions or validity sets), and (ii) exact or approximate solution functions defined on these regions or only the values of solution functions at specific points of the regions. Under some conditions, the computed exact functions or values provide respectively the (weakly) efficient functions or specific (weakly) efficient points making up the set $X_{(w)E}$ for the original MOQP($\boldsymbol{\theta}$), while approximate functions provide an approximation of this set.

Based on the state of the art in parametric optimization, we believe that the weighted-sum scalarization, epsilon-constraint scalarization, and their variations are the most useful for solving mpMOQPs. In the next section, therefore, we propose a generalized weighted sum scalarization for the nonparametric MOP and reduce it to a variety of SOPs whose optimal solutions are at least weakly efficient to the MOP. We then apply these scalarizations to mpMOQPs with suitable parametric optimization algorithms.

2.3 Scalarization Methods

ec:scalarizations

In this section, to keep the notation simple we depart from the parametric setting, and deal with a standard (nonparametric) MOP to prepare the ground for the algorithms we present in subsequent sections. We develop a weighted-sum scalarization encompassing several other scalarizing approaches. This scalarization employs several sets of parameters whose interplay allows for formulating variants of SOPs that will be useful for scalarizing mpMOQPs.

In this section only, let $g : \mathbb{R}^n \to \mathbb{R}^r$ denote the vector-valued objective function and let $\mathcal{X} \subseteq \mathbb{R}^n$ denote the feasible set. Then consider the MOP

$$\min_{\boldsymbol{x}\in\mathcal{X}} \; [g_1(\boldsymbol{x}),\; g_2(\boldsymbol{x}), \ldots, g_r(\boldsymbol{x})] \qquad \text{(MOP)}$$ mopBasic

and define the following sets and parameters for the index set $\{1, \ldots, r\}$.
We assume the MOP is (strictly) convex, that is, all objective functions $g_i(\boldsymbol{x})$, $i = 1, \ldots, r$ are (strictly) convex and the feasible set $\mathcal{X}$ is convex.

def:partition **Definition 19.** *Let $t \in \mathbb{N}$, $t \leq r$. Let the index set $\{1, \ldots, r\}$ be given. Define $t+1$ sets J, J_1, ..., J_t, where $J \subseteq \{1, \ldots, r\}$, $J_j \subseteq \{1, \ldots, r\}$ for $j = 1, \ldots, t$. Define the sets of parameters*

$$\Lambda := \left\{ \boldsymbol{\lambda} \in \mathbb{R}^{|J|} : \lambda_i \geq 0, i \in J, \sum_{i\in J} \lambda_i = 1 \right\}$$

and

$$\mathcal{M}^j := \left\{ \boldsymbol{\mu}^j \in \mathbb{R}^{|J_j|} : \mu_i^j \geq 0, i \in J_j, \sum_{i \in J_j} \mu_i^j = 1 \right\}$$

for all $j = 1, \ldots, t$. *For convenience, also define*

$$\boldsymbol{\mu} := \left[\boldsymbol{\mu}^1, \ldots, \boldsymbol{\mu}^t \right] \in \mathcal{M} := \mathcal{M}^1 \times \cdots \times \mathcal{M}^t.$$

Making use of Definition 19, consider another MOP in which every objective function is a weighted sum of the objective functions corresponding to each subset:

$$\min_{\boldsymbol{x} \in \mathcal{X}} \left[\sum_{i \in J} \lambda_i g_i(\boldsymbol{x}), \sum_{i \in J_1} \mu_i^1 g_i(\boldsymbol{x}), \ldots, \sum_{i \in J_t} \mu_i^t g_i(\boldsymbol{x}) \right] \tag{MOP'}$$ mopPrime

These two MOPs have the following relationship.

op:genweightedsum **Proposition 1.** *Let* (MOP) *be convex. Then* $\hat{\boldsymbol{x}} \in \mathcal{X}$ *is a weakly efficient solution to* (MOP) *if and only if* $\hat{\boldsymbol{x}}$ *is a weakly efficient solution to* (MOP') *for some* $\boldsymbol{\lambda} \in \Lambda$ *and* $\boldsymbol{\mu} \in \mathcal{M}$.

Proof. Let $\hat{\boldsymbol{x}} \in \mathcal{X}$ be a weakly efficient solution to (MOP). Then, by the classical result in [47], $\hat{\boldsymbol{x}}$ is an optimal solution to the SOP

$$\min_{\boldsymbol{x} \in \mathcal{X}} \sum_{i=1}^{r} \gamma_i g_i(\boldsymbol{x}) \tag{2.1}$$ ws

for some $\boldsymbol{\gamma} \in \Gamma$, where $\Gamma := \{\boldsymbol{\gamma} \in \mathbb{R}^r : \gamma_i \geq 0, i = 1, \ldots, r, \sum_{i=1}^r \gamma_i = 1\}$. Consider now the following weighted sum problem which derives from (MOP'):

$$\min_{\boldsymbol{x} \in \mathcal{X}} \rho_0 \sum_{i \in J} \lambda_i g_i(\boldsymbol{x}) + \rho_1 \sum_{i \in J_1} \mu_i^1 g_i(\boldsymbol{x}) + \cdots + \rho_t \sum_{i \in J_t} \mu_i^t g_i(\boldsymbol{x}), \tag{2.2}$$ wsPrime

where

$$\boldsymbol{\rho} := [\rho_0, \ \rho_1, \ \ldots, \ \rho_t] \in \mathcal{P} := \left\{ \boldsymbol{\rho} \in \mathbb{R}^{t+1} : \rho_i \geq 0, i = 1, \ \ldots, \ t+1, \sum_{i=0}^{t} \rho_i = 1 \right\}. \tag{2.3}$$ rhoDef

If one can find $\boldsymbol{\rho}$, $\boldsymbol{\lambda}$, and $\boldsymbol{\mu}$ such that $\hat{\boldsymbol{x}}$ is an optimal solution to (2.2), then, again by [47], $\hat{\boldsymbol{x}}$ will

equivalently be weakly efficient to (MOP'). Take

$$\rho_0 \lambda_i = \gamma_i, i \in J$$
$$\rho_j \mu_i^j = \gamma_i, i \in J_j, \quad j = 1, \ldots, t$$

which ensures that $\boldsymbol{\rho} \in \mathcal{P}$ as defined in (2.3). Then, $\hat{\boldsymbol{x}}$ is an optimal solution to (2.2), and equivalently, it is a weakly efficient solution to (MOP'). □

Depending on the definitions of the sets J and J_j, $j = 1, \ldots, t$, different variants of (MOP') can be formulated and different SOPs obtained, which may be useful depending on the needs of the optimization solver being used or the context of the decision situation being modeled. Consider the following six scalarizations, the first four of which are already established in the literature. The optimal solutions to these established scalarizations are known to be (weakly) efficient to (MOP') and, by Proposition 1, are also weakly efficient to (MOP). Because scalarizations (2.8) and (2.9) are new, in the subsequent propositions their relationships with (MOP) are examined.

cor:partition **Corollary 1.** *Let* (MOP) *and* (MOP') *be given, and the sets J and J_j, $j = 1, \ldots, t$ be defined as in Definition 19. Let $\mathcal{E}$ be a hypercube contained in a Euclidean space of the dimension as specified below.*

1. *Taking $J = \{1, \ldots, r\}$, $J_j = \emptyset$ converts* (MOP') *into the* ***weighted sum SOP*** *associated with* (MOP) *[47]:*

$$\min_{\boldsymbol{x} \in \mathcal{X}} \sum_{i=1}^{r} \lambda_i g_i(\boldsymbol{x}) \qquad (2.4)$$
$$s.t.\ \boldsymbol{\lambda} \in \Lambda \subseteq \mathbb{R}^r.$$

weightedsum

2. *Let $J = \{i\}$, $J_j = \{j\}$ for all $j = 1, \ldots, r, j \neq i$, $J_i = \emptyset$. Then the ϵ-constraint scalarization converts* (MOP'), *which is equivalent in this case to* (MOP), *into the* ***ϵ-constraint*** *$SOP_i, i =$*

$1, \ldots, r$, *associated with* (MOP) *[50]:*

$$\begin{aligned} &\min_{\boldsymbol{x} \in \mathcal{X}} g_i(\boldsymbol{x}) \\ &s.t.\ g_j(\boldsymbol{x}) \leq \epsilon_j,\ j = 1, \ldots, r, j \neq i \\ &\quad \boldsymbol{\epsilon} \in \mathcal{E} \subseteq \mathbb{R}^{r-1}. \end{aligned} \tag{2.5}$$ econstraint

3. *Let* $J = \{1, \ldots, r\}$, $J_j = \{j\}$ *for all* $j = 1, \ldots, r$. *Then the* ϵ*-constraint scalarization converts* (MOP') *into the* ***hybrid SOP*** *associated with* (MOP) *[49]:*

$$\begin{aligned} &\min_{\boldsymbol{x} \in \mathcal{X}} \sum_{i=1}^{r} \lambda_i g_i(\boldsymbol{x}) \\ &s.t.\ g_j(\boldsymbol{x}) \leq \epsilon_j,\ j = 1, \ldots, r \\ &\quad \boldsymbol{\lambda} \in \Lambda \subseteq \mathbb{R}^r,\ \boldsymbol{\epsilon} \in \mathcal{E} \subseteq \mathbb{R}^r. \end{aligned} \tag{2.6}$$ hybrid

4. *Let* $p \in \mathbb{N}$, $p < r$, $J \subset \{1, \ldots, r\}$, $|J| = p$, $J_j = \{j\}$ *for all* $j \notin J$, $J_j = \emptyset$ *for all* $j \in J$. *Then the* ϵ*-constraint scalarization converts* (MOP') *into the* ***modified hybrid SOP*** *associated with* (MOP) *[57]:*

$$\begin{aligned} &\min_{\boldsymbol{x} \in \mathcal{X}} \sum_{i \in J} \lambda_i g_i(\boldsymbol{x}) \\ &s.t.\ g_j(\boldsymbol{x}) \leq \epsilon_j,\ j \notin J \\ &\quad \boldsymbol{\lambda} \in \Lambda \subseteq \mathbb{R}^p,\ \boldsymbol{\epsilon} \in \mathcal{E} \subseteq \mathbb{R}^{r-p}. \end{aligned} \tag{2.7}$$ modhybrid

5. *If* $0 \leq t < r$, *and* J *and* J_j *are defined as in Definition 19, partitioning* $\{1, \ldots, r\}$, *then the* ϵ*-constraint scalarization converts* (MOP') *into a variant of the hybrid SOP called the* ***weighted hybrid SOP****:*

$$\begin{aligned} &\min_{\boldsymbol{x} \in \mathcal{X}} \sum_{i \in J} \lambda_i g_i(\boldsymbol{x}) \\ &s.t.\ \sum_{i \in J_j} \mu_i^j g_i(\boldsymbol{x}) \leq \epsilon_j,\ j = 1, \ldots, t \\ &\quad \boldsymbol{\lambda} \in \Lambda \subseteq \mathbb{R}^{|J|},\ \boldsymbol{\mu} \in \mathcal{M},\ \boldsymbol{\epsilon} \in \mathcal{E} \subseteq \mathbb{R}^t. \end{aligned} \tag{2.8}$$ whybrid

6. *Let $p \in \mathbb{N}$, $p < r$, $J \subset \{1, \ldots, r\}$, $|J| = p$, $J_j = \{j\}$ for all $j \notin J$, $J_j = \emptyset$ for all $j \in J$. Then the ϵ-constraint scalarization converts* (MOP') *into a variant of the ϵ-constraint SOP_i called the* ***reduced ϵ-constraint****, SOP_i, $i \notin J$:*

$$\begin{aligned} &\min_{\boldsymbol{x} \in \mathcal{X}} g_i(\boldsymbol{x}) \\ &s.t. \sum_{j \in J} \lambda_j g_j(\boldsymbol{x}) \leq \epsilon \\ &\quad g_j(\boldsymbol{x}) \leq \epsilon_j, \ j \notin J, \ j \neq i \\ &\quad \boldsymbol{\lambda} \in \Lambda \subseteq \mathbb{R}^p, \ \boldsymbol{\epsilon} \in \mathcal{E} \subseteq \mathbb{R}^{r-p}. \end{aligned} \tag{2.9}$$

redeconstraint

The weighted hybrid SOP (2.8) allows for weighing all objective functions in the new objective and constraints.

op:weightedhybrid

Proposition 2. *Let $\hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}(\boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{\epsilon})$ be an optimal solution to the weighted hybrid SOP* (2.8) *for some $\boldsymbol{\lambda} \in \Lambda$, $\boldsymbol{\mu} \in \mathcal{M}$, and $\boldsymbol{\epsilon} \in \mathcal{E}$. Then $\hat{\boldsymbol{x}}$ is a weakly efficient solution to* (MOP).

Proof. Let $\hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}(\boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{\epsilon})$ be an optimal solution to (2.8). Therefore, $\hat{\boldsymbol{x}}$ is feasible to (2.8), that is,

$$\sum_{i \in J_j} \mu_i^j g_i(\hat{\boldsymbol{x}}) \leq \epsilon_j, \ j = 1, \ldots, t. \tag{2.10}$$

proof11

Assume $\hat{\boldsymbol{x}} \notin \mathcal{X}_{wE}$. Then there exists a point $\bar{\boldsymbol{x}} \in \mathcal{X}$ such that

$$g_i(\bar{\boldsymbol{x}}) < g_i(\hat{\boldsymbol{x}}) \quad \text{for all} \ \ i = 1, \ldots, r. \tag{2.11}$$

proof33

Applying $\mu_i^j \geq 0$ not all 0, we have $\mu_i^j g_i(\bar{\boldsymbol{x}}) \leq \mu_i^j g_i(\hat{\boldsymbol{x}}), i \in J_j$ with at least one strict inequality, for $j = 1, \ldots, t$. Then

$$\sum_{i \in J_j} \mu_i^j g_i(\bar{\boldsymbol{x}}) < \sum_{i \in J_j} \mu_i^j g_i(\hat{\boldsymbol{x}})$$

for $j = 1, \ldots, t$, which makes $\bar{\boldsymbol{x}}$ feasible to (2.8) by (2.10). From (2.11), we obtain $\lambda_i g_i(\bar{\boldsymbol{x}}) \leq \lambda_i g_i(\hat{\boldsymbol{x}})$ for $i \in J$ with at least one strict inequality, and then

$$\sum_{i \in J} \lambda_i g_i(\bar{\boldsymbol{x}}) < \sum_{i \in J} \lambda_i g_i(\hat{\boldsymbol{x}}),$$

which contradicts the optimality of $\hat{\boldsymbol{x}}$. Therefore $\hat{\boldsymbol{x}}$ is weakly efficient to (MOP). □

The reduced ϵ-constraint scalarization (2.9) is motivated by the difficulty caused by the ϵ-constraint approach. When it is applied to (MOP), this method requires $r-1$ right-hand-side (rhs) values for the ϵ-constraints so that the resulting SOP is feasible. In (2.9), some or all of the $r-1$ ϵ-constraints are replaced with one constraint for which only one rhs value is needed, and therefore the resulting SOP is referred to as the reduced $\boldsymbol{\epsilon}$-constraint SOP_i, $i = 1, \ldots, r$.

prop:reduced **Proposition 3.** *If $\hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}(\boldsymbol{\lambda}, \boldsymbol{\epsilon})$ is an optimal solution to the reduced ϵ-constraint SOP_i (2.9) for some $\boldsymbol{\lambda} \in \Lambda, \boldsymbol{\epsilon} \in \mathcal{E}$ and some $i \notin J$, then $\hat{\boldsymbol{x}}$ is a weakly efficient solution to (MOP).*

Proof. The proof of Proposition 3, being similar to the proof of Proposition 2, is in Section 2.7. □

The final proposition in this section reveals that the modified hybrid and reduced ϵ-constraint SOPs jointly determine the efficiency of a feasible solution to a strictly convex (MOP).

Proposition 4. *Let (MOP) be strictly convex. A feasible solution $\hat{\boldsymbol{x}} \in \mathcal{X}$ is efficient to (MOP) if and only if $\hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}(\boldsymbol{\lambda}, \boldsymbol{\epsilon})$ is an optimal solution to the modified hybrid SOP (2.7) such that $g_j(\hat{\boldsymbol{x}}) = \epsilon_j, j \notin J$, and is also an optimal solution to the reduced ϵ-constraint SOP_i (2.9) such that $\sum_{j \in J} \lambda_j g_j(\hat{\boldsymbol{x}}) = \epsilon$ and $g_j(\hat{\boldsymbol{x}}) = \epsilon_j$, $j \notin J$, $j \neq i$, for some $\boldsymbol{\lambda} \in \Lambda \subseteq \mathbb{R}^p$ and $\boldsymbol{\epsilon} = (\epsilon, \epsilon_{j_1}, \ldots, \epsilon_{j_{r-p}}) \in \mathcal{E} \subseteq \mathbb{R}^{r-p+1}$, where $j_k \notin J$, $j \neq i$ for all $k = 1, \ldots, r-p$ for each $i \notin J$.*

Proof. From Corollary 7 in [57] , a solution $\hat{\boldsymbol{x}} \in \mathcal{X}$ is efficient to (MOP) if and only if $\hat{\boldsymbol{x}} = \boldsymbol{x}(\boldsymbol{\lambda})$ is efficient to $\min_{\boldsymbol{x} \in \mathcal{X}} \left(\sum_{j \in J} \lambda_j g_j(\boldsymbol{x}), g_{j_1}(\boldsymbol{x}), \ldots, g_{j_{r-p}}(\boldsymbol{x}) \right)$ for some $\boldsymbol{\lambda} \in \Lambda \subseteq \mathbb{R}^p$. Theorem 4.1 in [22] then yields the desired result. □

2.4 Parametric Quadratic Programs with Quadratic Constraints

ratic constraints

An application of the $\boldsymbol{\epsilon}$-constraint formulation (2.5) causes MOQP($\boldsymbol{\theta}$) to assume the form of a parametric quadratically constrained quadratic program (mpQCQP) formulated as

$$
\begin{aligned}
&\min_{\boldsymbol{x}} \; f_i(\boldsymbol{x};\boldsymbol{\theta}) = \tfrac{1}{2}\boldsymbol{x}^T Q_i(\boldsymbol{\theta})\boldsymbol{x} + \mathbf{p}_i(\boldsymbol{\theta})^T\boldsymbol{x} + c_i(\boldsymbol{\theta}) \\
&\text{s.t. } \boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta},\boldsymbol{\epsilon}) = \{\boldsymbol{x} \in \mathbb{R}^n : \qquad\qquad (\text{QCQP}(\boldsymbol{\theta},\boldsymbol{\epsilon}))
\end{aligned}
$$

mpmoqcqp

$$
\begin{aligned}
& f_j(\boldsymbol{x};\boldsymbol{\theta},\epsilon_j) = \tfrac{1}{2}\boldsymbol{x}^T Q_j(\boldsymbol{\theta})\boldsymbol{x} + \mathbf{p}_j(\boldsymbol{\theta})^T\boldsymbol{x} + c_j(\boldsymbol{\theta},\epsilon_j) \le 0 \quad j = 1,\dots,\tilde{r}, j \neq i, \\
& \tilde{A}(\boldsymbol{\theta})\boldsymbol{x} \leqq \tilde{\boldsymbol{b}}(\boldsymbol{\theta},\boldsymbol{\epsilon}), \\
& \boldsymbol{x} \geqq \mathbf{0}\} \\
& \boldsymbol{\theta} \in \Theta, \boldsymbol{\epsilon} \in \mathcal{E} \subseteq \mathbb{R}^{r-1},
\end{aligned}
$$

where $i \in \{1,\dots,\tilde{r}\}$, $\tilde{A} : \Theta \to \mathbb{R}^{\tilde{m}\times n}, \tilde{\boldsymbol{b}} : \Theta\times\mathcal{E} \to \mathbb{R}^{\tilde{m}}, \tilde{m} = m+r-\tilde{r}$. Recall that $\Theta \subseteq \mathbb{R}^\kappa$ is polyhedral as defined in Assumption 1. Scalarizing MOQP($\boldsymbol{\theta}$) into QCQP($\boldsymbol{\theta},\boldsymbol{\epsilon}$), the coefficients $c_j(\boldsymbol{\theta})$ have been redefined into $c_j(\boldsymbol{\theta},\epsilon_j) = c_j(\boldsymbol{\theta}) - \epsilon_j$ to account for the scalarizing parameter ϵ_j, $j = 1,\dots,\tilde{r}, j \neq i$, and the linear inequality constraints have been modified to include the $\boldsymbol{\epsilon}$-constraints of the form $\mathbf{p}_j(\boldsymbol{\theta})^T\boldsymbol{x} + c_j(\boldsymbol{\theta}) \le \epsilon_j$ for $j = \tilde{r}+1,\dots,r$, where $\boldsymbol{\epsilon} \in \mathcal{E} \subseteq \mathbb{R}^{r-1}$ is a new parameter introduced into the model so that the problem remains feasible.

Despite the fact that many algorithms have been developed to solve parametric nonlinear programs with parameters in various locations [85], we believe there is no algorithm that exactly solves mpQPs of type QCQP($\boldsymbol{\theta},\boldsymbol{\epsilon}$). A potential candidate could be the parametric quadratic approximation (mpQA) algorithm [59] which solves parametric nonlinear programs. Making use of quadratic approximation of the objective function and linear approximation of the constraints, this algorithm transforms the original problem into an mpQP that, however, does not allow parameters in the left-hand-side of the linear constraints. Parameters are included in the left-hand-side of the linear constraints in [99] but the functions only contain terms bilinear in $\boldsymbol{x}$ and parameters. Very recently, an algorithm for a class of mpQCQPs was proposed in [79], but it does not take into account parameters in the quadratic form of $\boldsymbol{x}$. Given the state-of-the art in parametric optimization, to solve QCQP($\boldsymbol{\theta},\boldsymbol{\epsilon}$) we choose a parametric approximate simplex (mpAS) method [14] that relies on

linear interpolation of all nonlinear functions. Since the numerical effectiveness of this method has not been examined, we intend to investigate the tradeoff between its simplicity and the quality of its obtained optimal solutions.

In addition to Assumption 1 that now holds for $\Theta \times \mathcal{E}$, we make the following assumptions about QCQP($\boldsymbol{\theta}, \boldsymbol{\epsilon}$) that are required by the mpAS method.

ass:quadratic

Assumption 2. *1. The dimension of the parameter space $\Theta \subseteq \mathbb{R}^\kappa$ is κ, that is, Θ is full-dimensional.*

2. The matrices $Q_i(\boldsymbol{\theta}), i = 1, \ldots \tilde{r}$, are positive semi-definite for all $\boldsymbol{\theta} \in \Theta$.

3. The elements in $Q_i, \boldsymbol{p}_i, c_i, i = 1, \ldots, \tilde{r}, \tilde{A}$, and $\tilde{\boldsymbol{b}}$ are convex functions of $\boldsymbol{\theta}$.

Since the last assumption is required for computation of the error between the true value of the objective function and its approximation, it might be relaxed if that error was defined differently. Note that the constraints are all affine with respect to $\boldsymbol{\epsilon}$ due to formulation (2.5), so there are no assumptions with respect to $\boldsymbol{\epsilon}$ other than feasibility.

2.4.1 The mpAS method

sec:ASMsect

The mpAS method is given in Algorithm 1. To incorporate the scalarization parameters, $\mathcal{E}$, the combined parameter space is redefined as $\Xi := \Theta \times \mathcal{E}$ such that $(\boldsymbol{\theta}, \boldsymbol{\epsilon}) = \boldsymbol{\xi} \in \Xi$. Using processing methods in MATLAB Multi-Parametric Toolbox [52], Ξ is first partitioned into a set of simplices Ξ_k, $k = 1, \ldots, s$. Note s is finite because Θ and $\mathcal{E}$ are assumed to be polyhedral. For a given simplex, Ξ_k, let $\boldsymbol{\zeta}_j^k$, $j = 1, \ldots, s_k$, be its vertices. For each $\boldsymbol{\zeta}_j^k$, QCQP($\boldsymbol{\theta}, \boldsymbol{\epsilon}$) is solved to get objective values $f_i(\boldsymbol{\zeta}_j^k)$ and vectors $\boldsymbol{x}(\zeta_j^k)$. These objective values and vectors are then used to compute linear interpolations of the optimal objective function, $\bar{f}(\boldsymbol{\theta}, \boldsymbol{\epsilon})$, and of the optimal solution function, $\bar{\boldsymbol{x}}(\boldsymbol{\theta}, \boldsymbol{\epsilon})$ to QCQP($\boldsymbol{\theta}, \boldsymbol{\epsilon}$). Treating $\boldsymbol{x}$ and $\boldsymbol{\xi}$ as variables, the maximum error between the approximation and the actual objective function is computed. If this error is too large or the split limit has not been reached, Ξ_k is split into smaller simplices and this process is repeated on each new simplex. At termination, the parameter space Ξ has been partitioned into simplices called invariancy regions and the associated approximate optimal objective function has been constructed as a piecewise linear function of the parameters. The approximate optimal solution functions are also available.

The mpAS algorithm proposed in [14] assumes that the program being solved is jointly convex in $\boldsymbol{x}, \boldsymbol{\theta}, \boldsymbol{\epsilon}$. In problem generation, we settled for mere biconvexity due to the scaling problems from forcing joint convexity. However, this relaxation does not adversely affect the performance of the algorithm.

alg:mopASM

Algorithm 1: Multiparametric Approximate Simplex Algorithm

Input : The initial polyhedron, $\Xi = \Theta \times \mathcal{E}$ over which to solve QCQP($\boldsymbol{\theta}, \boldsymbol{\epsilon}$), error tolerance, split limit

Output: piecewise function $\boldsymbol{x}^K(\boldsymbol{\xi})$ for some terminal $K \in \mathbb{N}$, and piecewise optimal function $f_i(\boldsymbol{x}^K; \boldsymbol{\xi})$

Step 0: Partition parameter space Ξ into a set of simplices Ξ_t, $t = 1, \ldots, s$, and set desirable tolerance and split limit. asmLine1

Step t: for each $t = 1, \ldots, s$

asmLine3 For each vertex $\boldsymbol{\zeta}^k$ of Ξ_t, compute vectors $\boldsymbol{x}(\boldsymbol{\zeta}^k)$ and objective values $f_i(\boldsymbol{\zeta}^k)$.

asmLine4 Compute linear interpolations $\bar{f}(\boldsymbol{\xi})$ and $\bar{\boldsymbol{x}}(\boldsymbol{\xi})$.

asmLine5 Compute error.

if *error within tolerance or split limit reached* **then**

| stop

else

asmLine9 Split Ξ_i into smaller simplices, which add to the set $\{1, \ldots, s\}$.

sec:numericASM

2.4.2 Numerical Results

Having not found significant benchmarking of the mpAS algorithm in the literature, we make our own to gauge general effectiveness. We have implemented mpAS in MATLAB building off of code found in [60] and applied it to randomly generated convex QCQPs with one and two parameters. To maintain consistency with the instances generated in Table 2.1, we only use up to five variables. The tests have been run on a Dell Inspiron 3153 with a 128 GB SSD (8 GB free), 8 GB RAM, running an Intel Core i3-6100U 2.30 GHz processor with two cores and four logical processors running Windows 10 Enterprise.

In both the results here and the applications in Section 2.5, we only split the parameter spaces five times in order to maintain useful visual distinctions in the invariancy regions. Table 2.2 and Table 2.3 contain the mean, median, and standard deviations of the time, approximation error, and number of simplices for a set of single-objective parametric QCQPs which take the form of problem QCQP($\boldsymbol{\theta}, \boldsymbol{\epsilon}$). Both tables share $n \in \{2, 3, 4, 5\}$, $\kappa \in \{1, 2\}$, $\mathcal{E} = \{0\}$; Table 2.2 applies to mp-QCQPs with $r = \tilde{r} = 2$, $m = 2$, while Table 2.3 applies to mp-QCQPs with $r = \tilde{r} = 1$, $m = 3$ to mimic the form of multiparametric Linear Complementarity Problems (mpLCPs) used

to generate Table 2.1. For $\kappa = 2$, the quadratic constraint and objective function are affine in $\boldsymbol{\theta}$. Note that $\epsilon \in \mathcal{E} = \{0\}$, so we henceforth refer only to $\boldsymbol{\theta}$. For $\kappa = 1$, we consider the unit interval $\Theta = [0, 1]$, while, for $\kappa = 2$, we consider a smaller rectangle. For $\kappa = 2$, poorly scaled instances were sometimes generated when data came from the same distribution (standard normal). To eliminate such issues, the size of the parameter space was altered for each n. We let $\Theta = [0.1, 0.9] \times [0.1, 0.9]$, for $n \in \{2, 3\}$, and $\Theta = [0.2, 0.6] \times [0.2, 0.6]$ for $n \in \{4, 5\}$.

Two MATLAB functions are utilized: *randn* randomly generates the coefficient matrices and vectors using a normal distribution, and *fmincon* performs the minimizations at the corresponding steps of Algorithm 1 using the SQP method with the default number of iterations. In addition, Multi-Parametric Toolbox (MPT), stated by its creators to be "a software tool for MATLAB," [52] is utilized to construct the simplices.

The generation of each instance is extended from [26]. Generating the quadratic constraint, $\boldsymbol{x}^T Q(\boldsymbol{\theta})\boldsymbol{x} + \mathbf{p}(\boldsymbol{\theta})^T \boldsymbol{x} + \boldsymbol{c}(\boldsymbol{\theta}) \leq \boldsymbol{b}(\boldsymbol{\theta})$, such that it is guaranteed to be feasible is as follows. If we assume functions of $\theta \in [0, 1]$ are affine in θ, then

$$\boldsymbol{c}(\theta) := \boldsymbol{c}^1 + \boldsymbol{c}^2\theta$$
$$\mathbf{p}(\theta) := \mathbf{p}^1 + \mathbf{p}^2\theta$$
$$Q(\theta) := Q^1 + Q^2\theta$$

We randomly generate $\mathbf{p}^1, \mathbf{p}^2 \in \mathbb{R}^n$ and $\boldsymbol{c}^1, \boldsymbol{c}^2 \in \mathbb{R}$. To choose the locations of θ in $\mathbf{p}(\theta)$, we generate a 0-1 matrix $R_{\mathbf{p}} \in \mathbb{R}^{n \times n}$ and $r_{\boldsymbol{c}} \in \{0, 1\}$. Then $\mathbf{p}(\theta) := \mathbf{p}^1 + \mathbf{p}^2 R_{\mathbf{p}}\theta$ and $\boldsymbol{c}(\theta) := \boldsymbol{c}^1 + \boldsymbol{c}^2 r_{\boldsymbol{c}}\theta$ are randomly chosen.

Guaranteeing the positive semi-definite nature of $Q(\theta)$ proceeds as follows. Let $P(\theta) := P_1 + P_2 R_P \theta$ with $P_1, P_2 \in \mathbb{R}^{n \times n}$ randomly generated in the same manner as for $\mathbf{p}(\theta)$ and $\boldsymbol{c}(\theta)$, with $R_P \in \mathbb{R}^{n \times n}$ being a randomly generated 0-1 matrix like $R_{\mathbf{p}}$. Then $Q(\theta) = P^T(\theta)P(\theta)$ is guaranteed to be positive semi-definite, but is not affine in θ. Therefore for $\kappa = 2$, we choose to guarantee the affine nature by replacing all appearances of θ^2 with a second parameter $\theta_2 \in [0, 1]$. The result is that $Q(\theta_1, \theta_2) = P^T(\theta_1)P(\theta_1)$, with θ_2 replacing θ_1^2, is now PSD and affine in $\boldsymbol{\theta} = (\theta_1, \theta_2)$. To ensure feasibility of the quadratic constraint, we select $\bar{\boldsymbol{b}}$ to be the vector of all ones and define $\boldsymbol{b}(\theta) := \bar{\boldsymbol{b}}^T Q(\boldsymbol{\theta})\bar{\boldsymbol{b}} + \mathbf{p}(\boldsymbol{\theta})^T \bar{\boldsymbol{b}} + \boldsymbol{c}(\boldsymbol{\theta})$. The linear constraint and objective function are generated in the same manner, with the linear constraint using the same chosen $\bar{\boldsymbol{b}}$; feasibility of the linear constraints

is ensured by using upper or lower bounds of $\boldsymbol{\theta}$ as necessary and defining $\boldsymbol{b}$ accordingly.

In Table 2.2, we observe that, as expected, the time increases with both the number of variables and the number of parameters. With the exception of $n = 4$, the time doubles for every additional variable, and more than doubles going from $\kappa = 1$ to $\kappa = 2$. A similar trend can be noticed in the number of simplices, which makes sense given that the algorithm stores information per simplex. We present the error as the average error per simplex and observe that, while there is a trending increase overall, a distinct pattern is far less noticeable. Similar results are observable in Table 2.3.

It is of interest to compare the runtimes between the spLCP and mpLCP methods in Table 2.1 and the mpAS method in Tables 2.2 and 2.3. Given the substantial difference in these times in favor of the mpAS method regardless of the dimension of the parameter space and the presence of a quadratic constraint, we consider the error values to be small enough to warrant using mpAS as an effective solution tool. In the next section we apply mpAS to some specific mpMOQPs to obtain insight into its application context that is complementary to the information obtained with the mpLCP method.

Table 2.1: Statistics for the CPU times (in seconds) for BOQPs scalarized with the weighted-sum method (2.4) and solved with the spLCP and mpLCP methods.[1]

tab:LCPresults

n	no. of instances	statistics	diagonal		diagonal+off-diagonal	
			spLCP	mpLCP	spLCP	mpLCP
2	10	mean	8.775	9.239	8.359	10.840
		median	7.682	7.954	8.157	9.800
		std. dev.	1.869	2.618	0.837	3.370
3	10	mean	9.270	9.885	35.159	42.269
		median	9.458	9.708	37.125	39.439
		std. dev.	0.582	3.086	5.353	24.258
4	10	mean	93.874	108.994	100.303	113.490
		median	98.614	106.501	101.267	110.448
		std. dev.	11.522	4.526	16.918	8.193
5	10	mean	147.006	159.543	225.630	223.276
		median	142.312	153.459	221.569	226.972
		std. dev.	12.022	4.137	45.509	62.452

[1] created by Dr. Jayasekara

Table 2.2: Statistics for the CPU times (in seconds) and objective function errors for parametric QCQPs with two linear constraints, one quadratic constraint, solved with the mpAS method.

tab:ASMresults

n	no. of instances	statistics	$\kappa = 1$			$\kappa = 2$		
			time	simplices	error	time	simplices	error
2	10	mean	0.448	8.7	0.005	1.496	20.1	0.007
		median	0.359	6	0.005	0.921	12	0.004
		std. dev.	0.308	6.308	0.001	1.223	16.045	0.009
3	10	mean	0.929	13.7	0.007	2.410	30.556	0.019
		median	0.547	7.5	0.005	1.104	14	0.005
		std. dev.	0.813	12.720	0.005	2.226	30.835	0.039
4	10	mean	1.734	23.7	0.028	2.212	25.7	0.009
		median	2.072	29	0.010	1.534	19	0.006
		std. dev.	0.744	9.417	0.036	1.785	21.161	0.007
5	10	mean	1.491	18.2	0.020	4.940	45.2	0.031
		median	1.069	16	0.001	5.216	46	0.023
		std. dev.	0.941	9.886	0.032	1.536	13.571	0.023

Table 2.3: Statistics for the CPU times (in seconds) and objective function errors for parametric QCQPs with three linear constraints, no quadratic constraint, solved with the mpAS method.

:ASMresultsNoQuad

n	no. of instances	statistics	$\kappa = 1$			$\kappa = 2$		
			time	simplices	error	time	simplices	error
2	10	mean	0.542	10.1	0.005	1.790	26.818	0.010
		median	0.399	8	0.004	1.135	19	0.006
		std. dev.	0.406	6.871	0.002	1.495	20.894	0.012
3	10	mean	0.852	13.3	0.007	2.791	39.9	0.012
		median	0.738	10.5	0.006	2.620	37.5	0.007
		std. dev.	0.619	9.696	0.004	1.564	22.078	0.015
4	10	mean	0.904	17.2	0.009	1.677	22.444	0.006
		median	0.936	16	0.006	1.527	20	0.005
		std. dev.	0.599	11.670	0.007	1.234	17.038	0.003
5	10	mean	1.372	22.9	0.008	2.844	35.200	0.010
		median	1.682	28.5	0.004	3.203	35	0.008
		std. dev.	0.750	12.179	0.007	1.795	23.380	0.008

2.5 Applications[1]

sec:application

In this section we apply the scalarizations presented in Section 2.3 and the mpLCP and mpAS algorithms to specific triobjective mpMOQPs resulting from applications in statistics and portfolio optimization. We apply all the scalarizations except for the hybrid SOP (2.6) and the weighted hybrid SOP (2.8). The latter needs to act on a problem with at least four objective functions to be meaningful. The former places all objective functions into both the objective and ϵ-constraints, which results in a numerically demanding method that seems not to provide beneficial

[1] models and data created by Dr. Jayasekara

information for the decision maker (DM).

sec:MOPelasticNet

2.5.1 The Elastic Net Problem

Consider a set of $n \in \mathbb{N}$ datapoints, stored as k-dimensional vectors; apply the elastic net statistical program, which minimizes the error by incorporating both the Euclidean norm and the absolute value norm into its minimization program, scaled by parameters α and β. These parameters are typically chosen heuristically, but we show that the presented methodology can enhance linear regression when the elastic net problem is solved to select regression parameters. Let the data set have n observations with k predictors. Consider the standard linear regression model in which the response $\mathbf{y} \in \mathbb{R}^k$ is predicted by

$$\hat{\mathbf{y}} = \Phi \boldsymbol{x}, \tag{2.12}$$

linmod

where $\Phi = \begin{bmatrix} \boldsymbol{\phi_1} & \boldsymbol{\phi_2} & \dots & \boldsymbol{\phi_n} \end{bmatrix} \in \mathbb{R}^{k \times n}$ is the design matrix with predictors $\boldsymbol{\phi_i} \in \mathbb{R}^k$ and $\boldsymbol{x} \in \mathbb{R}^n$ is the vector of "parameters" to be estimated. Here the word "parameters" has a different meaning than in parametric optimization. In statistics, the response and the predictors are both known, while the vector of parameters (typically denoted by $\hat{\beta}$) remains unknown and needs to be estimated so that the residual squared error is minimized. In the context of optimization, the unknown parameters become variables to be determined in the process of minimizing the squared error, which is modeled as the QP

$$\min_{\boldsymbol{x} \in \mathbb{R}^n} \quad f_1(\boldsymbol{x}) = \tfrac{1}{2} \boldsymbol{x}^T Q \boldsymbol{x} + \mathbf{p}^T \boldsymbol{x} + c, \tag{2.13}$$

statversion1

where $Q = 2\Phi^T \Phi$ is positive-definite, $\mathbf{p} = -2\Phi^T \mathbf{y}$ and $c = \mathbf{y}^T \mathbf{y}$. To avoid confusion, we refer to these parameters $\boldsymbol{x}$ as coefficients of (2.12). Because model (2.12) performs poorly in both prediction and interpretation when the estimated coefficients are computed from (2.13), this QP has been augmented by two penalty terms: the ℓ_2-term, known as the ridge penalty, and the ℓ_1-term, known as the lasso penalty. [101]. This leads to the elastic net problem.

$$\begin{aligned} \min_{\boldsymbol{x} \in \mathbb{R}^n} \quad & f_{\text{elastic-net}}(\boldsymbol{x}; \alpha, \beta) = \tfrac{1}{2} \boldsymbol{x}^T Q \boldsymbol{x} + \mathbf{p}^T \boldsymbol{x} + c + \tfrac{\alpha}{2} \boldsymbol{x}^T \boldsymbol{x} + \beta \textstyle\sum_{i=1}^n |x_i| \\ & \alpha, \beta \geq 0, \end{aligned} \tag{2.14}$$

elasticnet

where α, β play the role of modeling parameters (in agreement with the terminology introduced in Section 2.2). Because these parameters strongly affect the performance of model (2.12), they require

tuning.

In [20], the parameter tuning is modeled as a bilevel optimization problem involving minimization of the squared error on the efficient set of the triobjective QP (TOQP)

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \quad [f_1(\boldsymbol{x}) = \tfrac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + \mathbf{p}^T\boldsymbol{x} + c,\ f_2(\boldsymbol{x}) = \tfrac{1}{2}\boldsymbol{x}^T\boldsymbol{x},\ f_3(\boldsymbol{x}) = \textstyle\sum_{i=1}^n |x_i|], \tag{2.15}$$

TOQP

for which (2.14) is the associated weighted-sum SOP. Since the optimization over the efficient set remains challenging [96], an algorithm is proposed to compute the minimum squared error (minSE) with respect to continuously changing values of β but on a user-selected-grid of fixed values of α. Because of the grid, an optimal solution to the bilevel problem may not be achieved.

The methodology presented in this paper can further facilitate parameter tuning because (2.15) can now be solved for its complete parametric efficient set which then can be used to find α, β yielding the smallest squared error. We demonstrate this application on a simple example with $n = 3, k = 5$, the following data

$$\Phi = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 3 & 2 \\ 1 & 2 & 2 \\ 1 & 4 & 5 \\ 1 & 3.5 & 2.5 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 0.55 \\ 0.623 \\ 0.587 \\ 0.569 \\ 0.758 \end{bmatrix}, \tag{2.16}$$

example1

and by solving (2.15) in three ways.

2.5.1.1 The weighted-sum SOP

statWSSOP

Applying the weighted-sum (2.4) to (2.15) yields the following mpQP:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \quad & \lambda_1(\tfrac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + \mathbf{p}^T\boldsymbol{x} + c) + \tfrac{\lambda_2}{2}\boldsymbol{x}^T\boldsymbol{x} + (1-\lambda_1-\lambda_2)\textstyle\sum_{i=1}^n |x_i| \\ \lambda \in \Lambda', & \end{aligned} \tag{2.17}$$

weightedTOQP

where $\Lambda' = \{\boldsymbol{\lambda} \in \mathbb{R}^2 : \lambda_1, \lambda_2 \geq 0, \lambda_1 + \lambda_2 \leq 1\}$. The problem is reformulated to make all variables nonnegative which at the same time transforms f_3 into a linear function. Having solved (2.17) with the mpLCP method, we present the Pareto set in Figure 2.1a.

2.5.1.2 The ϵ-constraint SOP

The $\boldsymbol{\epsilon}$-constraint formulation may be more attractive to the DM than the weighted-sum (2.17) because the squared error, as the primary criterion, is directly minimized while the values of the secondary criteria are controlled by the scalarization parameters ϵ_2 and ϵ_3.

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \ & \tfrac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + \mathbf{p}^T\boldsymbol{x} + c \\ \text{s.t. } & \tfrac{1}{2}\boldsymbol{x}^T\boldsymbol{x} \le \epsilon_2 \\ & \sum_{i=1}^{3} |x_i| \le \epsilon_3 \\ & \boldsymbol{\epsilon} \in \mathcal{E}. \end{aligned} \tag{2.18}$$

epsConStatEx

By computing $\boldsymbol{x}^1 = \text{argmin}_{\boldsymbol{x}} f_1(\boldsymbol{x})$, we establish $\mathcal{E} = [0, f_2(\boldsymbol{x}^1) = 0.186] \times [0, f_3(\boldsymbol{x}^1) = 0.601]$. We solve (2.18) with the mpAS method and the obtained approximate optimal solutions, $\tilde{\boldsymbol{x}}(\boldsymbol{\epsilon})$, by Prop. 4.3 in [34], are approximate weakly efficient to (2.15). In Figure 2.1b, the approximate minSE, $\tilde{f}_1(\epsilon_2, \epsilon_3)$, is depicted, along with the Pareto points $(0.194, 0.605, 0.012)$, $(0.016, 0.203, 0.207)$, $(0.038, 0.195, 0.116)$, and $(0, 0, 1.933)$, labeled A, B, C, and D, respectively. Placing these Pareto points also in Figure 2.1a allows one to see that the approximation works well. Note that point A is beyond the unconstrained minimum of $f_1(\boldsymbol{x})$ and thus is not located on the weak Pareto set computed by applying the mpAS method to (2.18). The optimal solutions are available for every region in the partition.

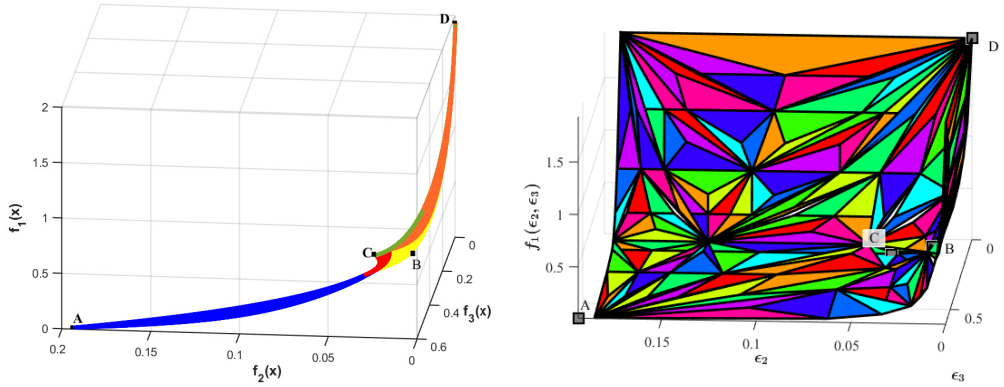
Consider the region containing $\bar{\boldsymbol{\epsilon}} = (0.119, 0.506)$,

$$IR = \left\{ \boldsymbol{\epsilon} \in \mathcal{E} : \begin{bmatrix} 0.853 & -0.502 \\ -0.709 & 0.655 \\ -0.940 & 0.334 \end{bmatrix} \begin{bmatrix} \epsilon_2 \\ \epsilon_3 \end{bmatrix} \le \begin{bmatrix} -0.144 \\ 0.262 \\ 0.072 \end{bmatrix} \right\}.$$

Then the optimal solution on this region is

$$\tilde{\boldsymbol{x}}(\boldsymbol{\epsilon}) = \begin{bmatrix} 1.539\epsilon_2 + 0.267\epsilon_3 - 0.041 \\ -1.002\epsilon_2 + 0.320\epsilon_3 + 0.123 \\ 0.537\epsilon_2 - 0.413\epsilon_3 + 0.083 \end{bmatrix}$$

[2] created by Dr. Jayasekara



(a) Pareto set for obtained by the mpLCP method on problem (2.17). (b) Approximate minSE computed by the mpAS method on problem (2.18).

fig:objSpaceEx2 fig:epsConStatEx fig:f1withparaEx2

Figure 2.1: The minSE for elastic net problem (2.15)

with the approximate minSE function $\tilde{f}_1(\boldsymbol{\epsilon}) = 0.188\epsilon_2 - 0.249\epsilon_3 + 0.126$. Note that $f_2(\bar{\boldsymbol{\epsilon}}) = 0.094$ and $f_3(\bar{\boldsymbol{\epsilon}}) = 0.456$, which result in the $\boldsymbol{\epsilon}$-constraints being approximately active.

2.5.1.3 The reduced $\boldsymbol{\epsilon}$-constraint SOP

Alternatively, the two secondary criteria may be combined into one constraint in the reduced $\boldsymbol{\epsilon}$-constraint formulation for (2.15), which results in the following mpQCQP

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \ & \tfrac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + \mathbf{p}^T\boldsymbol{x} + c \\ \text{s.t. } & \tfrac{1}{2}\lambda\boldsymbol{x}^T\boldsymbol{x} + (1-\lambda)\sum_{i=1}^{3}|x_i| \leq \epsilon \\ & \lambda \in [0,1], \epsilon \in \mathcal{E}. \end{aligned} \tag{2.19}$$

redEpsConStatEx

Problem (2.19) is presented in [101] and referred to as the naive elastic net problem, while the function $\frac{\lambda}{2}\lambda\boldsymbol{x}^T\boldsymbol{x} + (1-\lambda)\sum_{i=1}^{3}|x_i|$ is called the elastic net penalty. The authors of [101] solve this problem only for fixed values of the parameters λ and ϵ, while the approach presented here obtains optimal solutions for the entire parameter space.

Using $\boldsymbol{x}^1$, the unconstrained minimizer of f_1, we select $\mathcal{E} = [0, 0.6]$, as $\epsilon > 0.6$ results in f_1 attaining its global minimum, which is not of interest. Solving (2.19) with the mpAS method, we obtain the optimal solutions that, by Proposition 3, are weakly efficient to (2.15). The approximate minSE as a function of λ and ϵ is depicted in Figure 2.2. Note that ϵ constrains the λ-weighing of

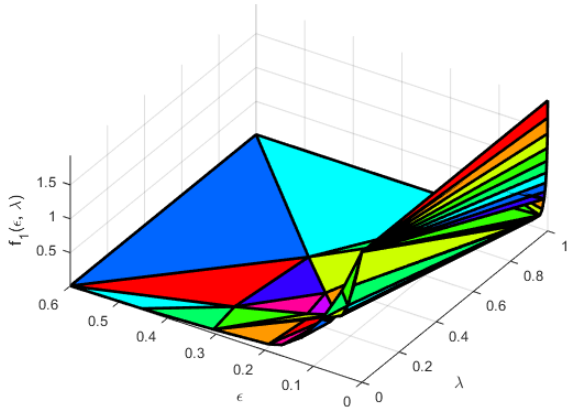


Figure 2.2: Approximate minSE for elastic net problem (2.15) obtained by the mpAS method on problem (2.19).

fig:f1ASMrEpsCon

f_2 and f_3.

As an example, consider the region containing $\bar{\lambda} = 0.955$, $\bar{\epsilon} = 0.087$,

$$IR = \left\{ (\lambda, \epsilon) : \begin{bmatrix} 0.707 & 0 \\ -0.105 & 0.994 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ \epsilon \end{bmatrix} \leq \begin{bmatrix} 0.707 \\ 0.0158 \\ -0.452 \end{bmatrix} \right\}.$$

Then the optimal solution on this region is

$$\tilde{\boldsymbol{x}}(\lambda, \epsilon) = \begin{bmatrix} 0.704\lambda + 2.235\epsilon - 0.682 \\ -0.096\lambda + 0.203\epsilon + 0.243 \\ -0.134\lambda - 0.892\epsilon + 0.169 \end{bmatrix}$$

with the approximate minSE function $\tilde{f}_1(\lambda, \epsilon) = -0.092\lambda - 0.880\epsilon + 0.215$. Using the reduced-$\epsilon$-constraint formulation, the DM can set $\bar{\lambda} = 0.955$ and $\bar{\epsilon} = 0.087$ to obtain the coefficients for fitting a curve to the data being $\tilde{\boldsymbol{x}}(\bar{\lambda}, \bar{\epsilon}) = (0.185, 0.169, -0.037)$. Based on how the approximated minSE value, $\tilde{f}_1(\bar{\lambda}, \bar{\epsilon}) = 0.04$, compares to other minSE values for other values of λ and ϵ, the DM can decide whether the obtained coefficients are appropriate.

sec:MOPportfolio

2.5.2 Portfolio Optimization

Consider the parametric triobjective portfolio optimization problem (TOPOP) with two quadratic objectives being the variances of portfolio return and liquidity and one parametric linear objective being the expected return,

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \quad & \left[f_1(\boldsymbol{x}) = \tfrac{1}{2}\boldsymbol{x}^T Q_1 \boldsymbol{x}, f_2(\boldsymbol{x}) = \tfrac{1}{2}\boldsymbol{x}^T Q_2 \boldsymbol{x}, f_3(\boldsymbol{x};\theta) = -\boldsymbol{p}_3(\theta)^T \boldsymbol{x} \right] \\ \text{s.t.} \quad & \mathbf{1}^T \boldsymbol{x} = 1 \\ & \boldsymbol{x} \geqq \mathbf{0}, \end{aligned} \tag{2.20}$$

example2

and with the following data

$$Q_1 = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 2 & 0 \\ -1 & 0 & 2.5 \end{bmatrix}, \qquad Q_2 = \begin{bmatrix} 3 & -1 & 0 \\ -1 & 4 & 1 \\ 0 & 1 & 3.5 \end{bmatrix}, \qquad \boldsymbol{p}_3(\theta) = \begin{bmatrix} -13.5 \\ 20 \\ \theta \end{bmatrix} \text{ for } \theta \in [15, 17].$$

A nonparametric version of this problem is solved in [57]. The modeling parameter θ represents the uncertain expected return on the capital invested in the third security. Solving this problem requires computing the set of (weakly) efficient solutions $\mathcal{X}_{(w)E} := \{X_{(w)E}(\theta)\}_{\theta\in[15,17]}$ and (weak) Pareto outcomes $\mathcal{Y}_{(w)P} := \{Y_{(w)P}(\theta)\}_{\theta\in[15,17]}$, as given in Definitions 17 and 18.

2.5.2.1 The modified-hybrid SOP

Problem (2.20) is reformulated with the modified hybrid method (2.7) which is particularly suited to this application because the risk functions are conveniently separated from the return function. The DM can weigh the overall risk by weighing the two risk functions with a parameter λ, while independently bounding from below the uncertain expected return (to be maximized) by another parameter ϵ. Additionally, the equality constraint is reformulated into two inequalities as required by the input to the mpLCP method. We obtain an mpQP in the form of an mpLCP with the modeling parameter θ and two parameters λ and ϵ which, despite their scalarizing role, fit the context of this application very well:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \quad & \tfrac{1}{2}\boldsymbol{x}^T Q(\lambda)\boldsymbol{x} \\ \text{s.t.} \quad & \tilde{A}(\theta)\boldsymbol{x} \leqq \tilde{\boldsymbol{b}}(\epsilon) \\ & \boldsymbol{x} \geqq \boldsymbol{0} \\ & \theta \in \Theta = [15,17], \lambda \in \Lambda' = [0,1], \epsilon \in \mathcal{E} = [-20, 13.5], \end{aligned} \tag{2.21}$$

modhybport

where $Q(\lambda) = \lambda Q_1 + (1-\lambda)Q_2$, and

$$Q(\lambda) = \begin{bmatrix} 3-2\lambda & \lambda-1 & -\lambda \\ \lambda-1 & 4-2\lambda & 1-\lambda \\ -\lambda & 1-\lambda & 7/2-\lambda \end{bmatrix}, \tilde{A}(\theta) = \begin{bmatrix} 13.5 & -20 & -\theta \\ 1 & 1 & 1 \\ -1 & -1 & -1 \end{bmatrix}, \tilde{\boldsymbol{b}}(\epsilon) = \begin{bmatrix} \epsilon \\ 1 \\ -1 \end{bmatrix}.$$

The parameter intervals Θ and $\mathcal{E}$ are normalized. Applying $\theta^{nor} = \frac{\theta-\theta^{min}}{\theta^{max}-\theta^{min}} = \frac{\theta-15}{17-15}$ we have $\Theta^{nor} = [0,1]$. Applying $\epsilon^{nor} = \frac{\epsilon-\epsilon^{min}}{\epsilon^{max}-\epsilon^{min}} = \frac{\epsilon+20}{13.5-(-20)}$, where $\epsilon^{min} = \min\{-\mathbf{p}(\theta)_3^T\boldsymbol{x} : \mathbf{1}^T\boldsymbol{x} = 1, \boldsymbol{x} \geqq \boldsymbol{0}, \theta \in [15,17]\} = -20$ and $\epsilon^{max} = \max\{-\mathbf{p}_3(\theta)^T\boldsymbol{x} : \mathbf{1}^T\boldsymbol{x} = 1, \boldsymbol{x} \geqq \boldsymbol{0}, \theta \in [15,17]\} = 13.5$, we have $\mathcal{E}^{nor} = [0,1]$.

We also solve (2.21) with the mpAS method to compare its performance against the mpLCP method. Since the mpAS method allows equality or inequality constraints, we solve the following problem:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \ & \tfrac{1}{2}\boldsymbol{x}^T Q(\lambda)\boldsymbol{x} \\ \text{s.t. } & 13.5x_1 - 20x_2 - \theta x_3 \leq \epsilon \\ & \mathbf{1}^T\boldsymbol{x} = 1 \\ & \boldsymbol{x} \geqq \boldsymbol{0} \\ & \theta \in \Theta,\ \lambda \in [0,1],\ \epsilon \in \mathcal{E}. \end{aligned} \tag{2.22}$$

modhybportASM

where $Q(\lambda)$, Θ, and $\mathcal{E}$ are defined in (2.21), with Θ and $\mathcal{E}$ normalized.

Due to the fact that the approximations of $\boldsymbol{x}$ are in terms of λ, ϵ, and θ, to readily compare the solution of problem (2.21) to the solution of problem (2.22), we consider the vertices, $(\bar{\lambda}, \bar{\epsilon})$, of the invariancy regions containing $\bar{\theta} = 16$. Figure 2.3 contains the Pareto set for $\theta = 16$ overlaid with a graph of the return $f_3(\boldsymbol{x}(\bar{\lambda}, \bar{\epsilon}, 16))$ against the risks $f_1(\boldsymbol{x}(\bar{\lambda}, \bar{\epsilon}, 16))$ and $f_2(\boldsymbol{x}(\bar{\lambda}, \bar{\epsilon}, 16))$. Note that,

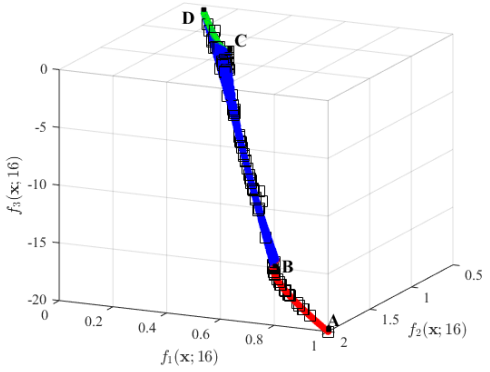


Figure 2.3: Approximate Pareto points for (2.20) and $\theta = 16$ computed by the mpAS method on problem (2.22).

bPortfolioTheta16

while the mpLCP method provides a complete description of the Pareto set, the mpAS method is capable of providing a good approximation on its own.

2.5.2.2 The ϵ-constraint SOP

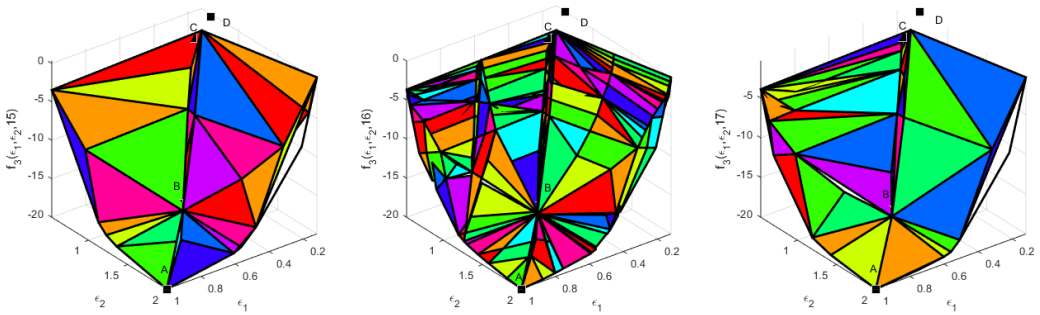
The $\boldsymbol{\epsilon}$-constraint formulation allows us to treat the return maximization as the primary objective while the values of each risk are controlled parametrically.

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \ & -\boldsymbol{p}_3(\theta)^T\boldsymbol{x} \\ \text{s.t. } & \tfrac{1}{2}\boldsymbol{x}^T Q_1 \boldsymbol{x} \le \epsilon_1 \\ & \tfrac{1}{2}\boldsymbol{x}^T Q_2 \boldsymbol{x} \le \epsilon_2 \\ & \mathbf{1}^T\boldsymbol{x} = 1 \\ & \boldsymbol{x} \geqq \mathbf{0} \\ & \theta \in \Theta,\ \boldsymbol{\epsilon} \in \mathcal{E}. \end{aligned} \tag{2.23}$$

epsConPortEx

Note that $\mathcal{E} = [\epsilon_1^{\min}, \epsilon_2^{\min}] \times [\epsilon_1^{\max}, \epsilon_2^{\max}]$, where $\epsilon_1^{\min}$, $\epsilon_1^{\max}$, $\epsilon_2^{\min}$, and $\epsilon_2^{\max}$, are computed by individually optimizing each criterion in problem (2.20) using $\theta = 16$. Then $\epsilon_1^{\min} = 0.120$ and $\epsilon_2^{\min} = 0.530$, while $\epsilon_1^{\max} = \max\{f_1(\hat{x}_2), f_1(\hat{x}_3)\} = 1$, $\epsilon_2^{\max} = \max\{f_2(\hat{x}_1), f_2(\hat{x}_3)\} = 2$, where $\hat{x}_i$ is the minimizer of f_i. The mpAS method is then applied to obtain approximate solutions.

The graphs in Figure 2.4 depict the weak Pareto sets for problem (2.20) [50]. Note that ϵ_1 constrains f_1 and ϵ_2 constrains f_2. The figures have been rotated to show reference points A,



psConPortfolioTheta15 fig:epsConPortfolioTheta16 fig:epsConPortfolioTheta17

(a) $\theta = 15$. (b) $\theta = 16$. (c) $\theta = 17$.

Figure 2.4: Approximate weak Pareto sets for (2.20) obtained by the mpAS method on problem (2.23).

SMepsConPortfolio

B, C, and D, and the narrow invariancy regions caused by numerical discrepancies are included in Figures 2.4a and 2.4c for completion. Note that points A, B, and C are included in the obtained approximate solutions, while D, also a Pareto point, is not. Nevertheless, this example demonstrates that the mpAS method provides relevant information and can be applied if necessary.

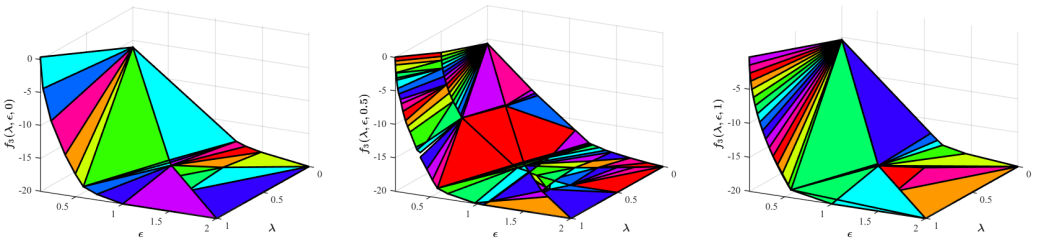
By holding $\bar{\theta}$ constant, the DM can readily see how limiting the risk with various values $\bar{\boldsymbol{\epsilon}}$ affects the return. Suppose that the return is $\bar{\theta} = 17$, and the DM wishes to limit the risks with $\bar{\boldsymbol{\epsilon}} = (0.517, 1.445)$. Then the encapsulating invariancy region is

$$IR = \left\{ (\epsilon_1, \epsilon_2, \theta^{nor}) : \begin{bmatrix} 0.774 & 0.155 & -0.463 \\ -0.569 & -0.340 & -0 \\ -0.822 & 0.292 & 0.312 \\ 0.6 & 0 & 0.528 \end{bmatrix} \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \theta^{nor} \end{bmatrix} \leq \begin{bmatrix} 0.403 \\ -0.749 \\ 0.379 \\ 0.600 \end{bmatrix} \right\}.$$

The optimal solution on this region,

$$\tilde{\boldsymbol{x}}(\epsilon_1, \epsilon_2, \theta^{nor}) = \begin{bmatrix} -0.789\epsilon_1 - 0.011\epsilon_2 - 0.059\theta^{nor} + 0.489 \\ 0.731\epsilon_1 - 0.011\epsilon_2 + 0.016\theta^{nor} + 0.212 \\ 0.05\epsilon_1 + 0.0219\epsilon_2 + 0.043\theta^{nor} + 0.299 \end{bmatrix}$$

has the approximate negative return function $\tilde{f}_3(\boldsymbol{\epsilon}, \theta^{nor}) = -26.173\epsilon_1 - 0.290\epsilon_2 - 2.545\theta - 2.069$. Then the amounts to be invested are found by the mpAS method to be $\tilde{\boldsymbol{x}}(0.517, 1.445, 1) = (0.036, 0.583, 0.382)$ yielding a negative return $\tilde{f}_3(0.517, 1.445, 1) = -17.590$, which is a gain of



(a) $\theta = 15$. (b) $\theta = 16$. (c) $\theta = 17$.

psConPortfolioTheta15 | fig:redEpsConPortfolioTheta16 | fig:redEpsConPortfolioTheta17

edEpsConPortfolio

Figure 2.5: Approximate optimal return obtained by the mpAS method on problem (2.24).

17.590.

2.5.2.3 The reduced $\boldsymbol{\epsilon}$-constraint SOP

We conclude by considering one of the new SOP formulations, the reduced $\boldsymbol{\epsilon}$-constraint SOP, to which we apply the mpAS method. In the reduced $\boldsymbol{\epsilon}$-constraint formulation, the return is still maximized while the parametrized weighted-sum of both risk functions is parametrically controlled.

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^3} \ & -\boldsymbol{p}_3(\theta)^T\boldsymbol{x} \\ \text{s.t. } & \tfrac{1}{2}\lambda\boldsymbol{x}^T Q_1\boldsymbol{x} + \tfrac{1}{2}(1-\lambda)\boldsymbol{x}^T Q_2\boldsymbol{x} \leq \epsilon \\ & \mathbf{1}^T\boldsymbol{x} = 1 \\ & \boldsymbol{x} \geqq \mathbf{0} \\ & \theta \in \Theta, \lambda \in [0,1], \epsilon \in \mathcal{E}. \end{aligned} \tag{2.24}$$

redEpsConPortEx

Note that solving the $\boldsymbol{\epsilon}$-constraint formulation gives the weak Pareto set as seen in Figure 2.4, but the weak Pareto set is not readily available from solving the reduced-$\boldsymbol{\epsilon}$-constraint formulation. As a result, we instead provide Figure 2.5. We note that as we fix $\bar{\theta}$, adjusting the weighting of the risks (the value of λ) affects the return given a total risk bound $\bar{\epsilon}$, as is expected. Likewise notice that Figure 2.5 demonstrates that, for fixed $\bar{\lambda}$, decreasing the total allowable risk, ϵ, will result in a lower return.

Consider the region containing $\bar{\lambda} = 0.613$, $\bar{\epsilon} = 0.639$, $\bar{\theta} = 0.5$ corresponding to a return

Table 2.4: CPU times (in seconds) for the scalarizations of Elastic Net and Portfolio problems solved with the mpLCP[4] and mpAS methods.

tab:summaryTable

Solution Method	Scalarization	Elastic net time	Scalarization	Portfolio time
mpLCP	weighted-sum (not shown)	91.673	modified hybrid (2.21)	67.367
mpAS	ϵ-constraint (2.18)	18.941	modified hybrid (2.22)	14.546
	reduced ϵ-constraint (2.19)	10.937	ϵ-constraint (2.23)	39.821
			reduced ϵ-constraint (2.24)	19.615

value of 16. Then the encapsulating invariancy region is

$$IR = \left\{ (\lambda, \epsilon, \theta^{nor}) : \begin{bmatrix} 0 & 0.469 & -0.881 \\ 0.415 & -0.903 & -0.014 \\ -0.707 & 0 & 0.707 \\ 0.373 & 0.590 & 0.009 \end{bmatrix} \begin{bmatrix} \lambda \\ \epsilon \\ \theta^{nor} \end{bmatrix} \leq \begin{bmatrix} 0.056 \\ -0.108 \\ 0 \\ 0.716 \end{bmatrix} \right\}.$$

The optimal solution on this region,

$$\tilde{\boldsymbol{x}}(\lambda, \epsilon, \theta^{nor}) = \begin{bmatrix} -0.179\lambda - 0.283\epsilon - 0.004\theta + 0.343 \\ 0.245\lambda + 0.184\epsilon - 0.040\theta + 0.310 \\ 0.005\lambda + 0.212\epsilon + 0.046\theta + 0.210 \end{bmatrix}$$

has the approximate negative return function $\tilde{f}_3(\lambda, \epsilon, \theta^{nor}) = -7.336\lambda - 10.738\epsilon - 0.733\theta^{nor} - 4.725$. Then $\tilde{\boldsymbol{x}}(\bar{\lambda}, \bar{\epsilon}, \bar{\theta}) = (0.050, 0.558, 0.372)$ yields $\tilde{f}_3(\bar{\lambda}, \bar{\epsilon}, \bar{\theta}) = -16.451$, which is a gain of 16.451.

2.5.3 Summary

In this section we solved two triobjective optimization problems using four scalarizations and two solution algorithms resulting in seven cases. We have recognized a very good quality of approximate efficient solutions provided by the mpAS method in comparison with the efficient solutions provided by the mpLCP method. In Table 2.4 we compare the CPU times for all seven cases and observe that the mpAS method decisively outperforms the mpLCP method. Facing a tradeoff between the CPU time and the solution quality, we conclude that the mpAS method can serve as an attractive tool to the DMs who consider runtime to be their most important resource, while, for situations requiring more precision, the mpLCP method may be the better option.

[4] run by Dr. Jayasekara

sec:conclusion

2.6 Conclusion

This paper appears to present the first numerical study on solving parametric MOPs. We showed that the state-of-the art parametric optimization algorithms allow computation of efficient sets for convex mpMOQPs in which parameters model unknown or uncertain quantities. mpMOQPs are solved by scalarization which introduces additional parameters when transforming the original problem into mpQPs. Because the efficient set for mpMOQPs is a parametrized collection of the efficient sets, the former can be computed by the algorithms that have been designed for mpQPs as long as they are able to handle multiple parameters and work well on mpQPs of different types. To offer flexibility with mpQPs, we proposed a generalized weighted-sum scalarization that reduces to six SOPs of the weighted-sum/epsilon-constraint type that can be applied depending on the real-life context and available solver.

In addition to work not portrayed here involving the mpLCP method, we experimented with applying the mpAS method to single objective mpQCQPs. The mpAS method cannot find the true Pareto sets as well as the mpLCP method but is capable of approximating (weak) Pareto sets in totality and providing tradeoff information that is different from that offered by the mpLCP method. In the absence of an exact method, the mpAS method turns out to be an effective solution tool as the only method to approximately solve mpQCQPs resulting from scalarizing mpMOQPs. Furthermore, mpAS can also be a useful method when the mpLCP method is applicable but not of interest because a short computational time is more important than the solution exactness. Using the elastic net problem in statistics and a portfolio optimization problem, we showed that matching the two methods with specific scalarizations provides mutually complementary insight into the real-life context and therefore supports decision making.

Future research can go in different directions. Software advances in solving systems of polynomial equations might make the spLCP method more efficient in the future. The mpLCP method could become more competitive for BOQPs if its implementation was reduced for single-parameter LCPs. There is a need to develop algorithms for mpQCQPs since they emerge from useful epsilon-constraint-type scalarizations of mpMOQPs. The mpAS method, that relies on joint convexity, could be extended to apply to biconvexity to allow for broader applicability. Dealing with nonconvex problems is also important. Dropping the convexity assumption and making use of available convex relaxations for nonconvex QPs may allow application of the presented algorithms

to nonconvex MOQPs and, consequently, nonconvex mpMOQPs. More generally, since parametric multiobjective optimization requires methodologies that are customized at the stage of SOP reformulation and the stage of algorithmic development, it is advisable to design algorithms for specific classes of mpSOPs and utilize them for mpMOPs.

2.7 Proof of Proposition 3

proofpendix

Consider, again, Proposition 3.

Proposition 5. *If $\hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}(\boldsymbol{\lambda}, \boldsymbol{\epsilon})$ is an optimal solution to the reduced ϵ-constraint SOP$_i$ (2.9) for some $\boldsymbol{\lambda} \in \Lambda, \boldsymbol{\epsilon} \in \mathcal{E}$ and some $i \notin J$, then $\hat{\boldsymbol{x}}$ is a weakly efficient solution to (MOP).*

Proof. Let $\hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}(\boldsymbol{\lambda}, \boldsymbol{\epsilon})$ be an optimal solution to (2.9) for some $i \notin J$. Therefore, $\hat{\boldsymbol{x}}$ is feasible to (2.9), that is,

$$\sum_{j \in J} \lambda_j g_j(\hat{\boldsymbol{x}}) \leq \epsilon \tag{2.25}$$ proof1

$$g_j(\hat{\boldsymbol{x}}) \leq \epsilon_j \quad j \notin J,\ j \neq i. \tag{2.26}$$ proof2

Assume $\hat{\boldsymbol{x}} \notin \mathcal{X}_{wE}$. Then there exists a point $\bar{\boldsymbol{x}} \in \mathcal{X}$ such that

$$g_j(\bar{\boldsymbol{x}}) < g_j(\hat{\boldsymbol{x}}) \quad \text{for all} \quad j = 1, \ldots, r. \tag{2.27}$$ proof3

Applying $\lambda_j \geq 0$ not all 0, we have $\lambda_j g_j(\bar{\boldsymbol{x}}) \leq \lambda_j g_j(\hat{\boldsymbol{x}}),\ \ j \in J$ with at least one strict inequality; therefore $\sum_{j \in J} \lambda_j g_j(\bar{\boldsymbol{x}}) < \sum_{j \in J} \lambda_j g_j(\hat{\boldsymbol{x}})$. Using (2.25), we obtain

$$\sum_{j \in J} \lambda_j g_j(\bar{\boldsymbol{x}}) < \epsilon, \tag{2.28}$$ proof4

while from (2.26) and (2.27),

$$g_j(\bar{\boldsymbol{x}}) < \epsilon_j \quad j \notin J,\ j \neq i. \tag{2.29}$$ proof5

Since $\bar{\boldsymbol{x}} \in \mathcal{X}$, (2.28) and (2.29) make $\bar{\boldsymbol{x}}$ feasible to (2.9). From (2.27), $g_i(\bar{\boldsymbol{x}}) < g_i(\hat{\boldsymbol{x}})$, which contradicts the optimality of $\hat{\boldsymbol{x}}$. Therefore, $\hat{\boldsymbol{x}}$ is weakly efficient to (MOP). □

Chapter 3

Approximating Optimal Solutions to Biconvex Parametric Programs

ch:BASM

[This chapter consists of a paper submitted for review to the journal *Optimization Letters* in April 2022; it is currently under revision as [77].]

Many optimization programs are motivated by situations which culminate, among other things, from uncertainty—from unknown or imprecise data, inaccurate measurements, or potentially inadequate models. In the operations research literature, there are three classical paradigms to model uncertainty: probabilistic, possibilistic, and deterministic. We consider the deterministic approach by defining domains containing the possible uncertainties. This approach has given foundation to parametric optimization [42] and robust optimization [15].

In addition to the constants and variables in all mathematical programs, a parametric program also contains exogeneous variables, called parameters which model uncertainty because their values are not being solved for, but are not known at the time of solving. In contrast with the nonparametric program, which is solved to obtain a specific solution vector and corresponding optimal value, the parametric program is solved to obtain a solution vector-valued function and the corresponding optimal value function, both of which map the parameters to their corresponding solution spaces. The form of these solutions fundamentally changes the problem.

Studies on parametric programming go back at least to 1964 when the authors of [66] provide properties of solution functions based on the structure of the program. Since then, researchers have

worked on the theory, summarized in [42] and [43]. A basic penalty function algorithm is given in [42], with more algorithmic methods [29, 31, 33, 84] coming about with the turn of the millenium. These authors work with parameters only in the right-hand side of the constraints, but in 2006, the authors of [14] present a more versatile approximation approach, which we term the Approximate Simplex Method (ASM). The ASM allows parameters to be located, under certain conditions, in both the objective and the constraint functions. In 2013, [28] incorporates parametric objective functions into a mixed integer program with parameters only in the right-hand side, emulating [31].

More recently, work has been done to support parameters in other locations: [86] and [80] solve multiparametric programs at most bilinear in the parameter and variable, quadratic in the variable, and quadratic in the parameter for both the objective and constraint functions; [5] posits the first method for solving parametric convex quadratic programs with parameters in general locations; finally, [17] and [23] apply Gröbner bases to solve the first order KKT conditions of multiparametric programs with parameters anywhere in nonlinear objective and constraint functions. Also worth noting is the use of finite element methods for solving partial differential equations to solve optimization programs by solving the KKT systems, such as in [9] and [86].

In addition to uncertainty of coefficients, real-life programs frequently involve multiple objective functions; the solving of multiple objective programs relies heavily on parametric single objective programs. Due to the versatility of the algorithm which [14] provides, it appears in [58] as part of the process to solve multi-objective programs via scalarization. In that application it is found that, while the scalarizing single-objective programs do not meet the conditions put forth in [14], the ASM still succeeds on the programs examined.

To explain this unexpected success, we consult the literature, but, as detailed above, there are not any studies on the solving of parametric programs involving objectives and constraints biconvex in the variables and parameters beyond the works [80] and [86]. The ability to solve programs of this nature opens up multiple scalarization methods in multiobjective programs and the application of gradient dual ascent to single-objective programs. As a result, in this work, we extend the ASM to apply to a broader set of objectives and constraints than originally posited by [14]; we supplement the presented theory with a brief numerical exploration of a set of randomly generated examples.

We begin by discussing the biconvex parametric program being solved in Section 3.0.1 along with the definitions in use, and we summarize the algorithm from [14] in Section 3.0.2. We provide

the theory to extend the ASM from [14] to a broader set of objectives and constraints in Section 3.1; specifically, we examine the feasibility of the solution function in Section 3.1.1, and the new termination conditions in Section 3.1.2. Finally, in Section 3.2, we construct the algorithm and compare both termination methods when applied to a set of randomly generated examples.

sec:defn

3.0.1 Definitions and Problem Statement

In this paper, Θ represents the space from which the parameters are drawn. In the case of some theorems, convexity of Θ may be sufficient, but for the algorithms presented, polygonality of Θ is necessary.

Let $m, n, \kappa \in \mathbb{N}$, $\kappa \geq 2$, $\Theta \subseteq \mathbb{R}^{\kappa-1}$ be a bounded (convex) polyhedron, $\boldsymbol{\theta} \in \Theta$ represent the parameters, and $\boldsymbol{x} \in \mathbb{R}^n$ represent the decision variables.

Letting $f : \mathbb{R}^n \times \Theta \to \mathbb{R}$ and $g_i : \mathbb{R}^n \times \Theta \to \mathbb{R}$, $i \in \{1, \ldots, m\}$, be biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$ and twice differentiable in $(\boldsymbol{x}, \boldsymbol{\theta})$, we consider the optimal value function $f^* : \Theta \to \mathbb{R}$ defined by the following biconvex program

$$\begin{aligned} f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathbb{R}^n_{\geqq}} \; & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ \text{s.t. } & g_i(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0, \; i \in \{1, \ldots, m\} \\ & A\boldsymbol{x} + B\boldsymbol{\theta} + \boldsymbol{d} = \boldsymbol{0}. \end{aligned} \tag{3.1}$$

bicon

If we let

$$\mathcal{X}(\boldsymbol{\theta}) := \{\boldsymbol{x} \in \mathbb{R}^n_{\geqq} : g_i(\boldsymbol{x}, \boldsymbol{\theta}) \leq 0, \; i = 1, \ldots, m, \; A\boldsymbol{x} + B\boldsymbol{\theta} + \boldsymbol{d} = \boldsymbol{0}\}$$

and $\mathcal{X}(\Theta) := \cup_{\boldsymbol{\theta} \in \Theta} \mathcal{X}(\boldsymbol{\theta})$, then we have the following definition. Note that $\boldsymbol{x}$ in (3.1) and $\boldsymbol{x}^i$, $i \in \{1, \ldots, \kappa\}$, in Definition 20 are vectors in $\mathbb{R}^n$, but $\boldsymbol{x}^*$ and $\bar{\boldsymbol{x}}$ below are vector-valued functions of $\boldsymbol{\theta}$.

solnDef **Definition 20.** *1. The **optimal solution function** to* (3.1)*, $\boldsymbol{x}^* : \Theta \to \mathbb{R}^n$, is defined by*

$$\boldsymbol{x}^* : \boldsymbol{\theta} \mapsto argmin_{\boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})} f(\boldsymbol{x}, \boldsymbol{\theta})$$

and the ***optimal value function****,* $f^*(\boldsymbol{\theta}) : \Theta \to \mathbb{R}$*, is defined by*

$$f^* : \boldsymbol{\theta} \mapsto f(\boldsymbol{x}^*(\boldsymbol{\theta}), \boldsymbol{\theta}).$$

2. *Define* $\bar{\boldsymbol{x}} : \Theta \to \mathbb{R}^n$ *and* $\bar{f} : \Theta \to \mathbb{R}$ *to be the* ***linear interpolants*** *of* $\boldsymbol{x}^*$ *and* f^* *respectively.*

3. *If* $\boldsymbol{\vartheta}^i$ *is a vertex of* Θ*,* $i \in \{1, \ldots, \kappa\}$*, then, for all* i*, the* ***vertex solution*** $\boldsymbol{x}^i := \boldsymbol{x}^*(\boldsymbol{\vartheta}^i)$ *is an optimal solution of program* (3.1) *at* $\boldsymbol{\vartheta}^i$*; that is,* $f^*(\boldsymbol{\vartheta}^i) = f(\boldsymbol{x}^i, \boldsymbol{\vartheta}^i)$*.*

Two generalizations of the convexity of a function are defined below.

biconDef **Definition 21.** *Let* $\lambda_1, \lambda_2 \in [0, 1]$*; consider convex spaces* $\mathcal{X}(\Theta) \subseteq \mathbb{R}^n$ *and* $\Theta \subseteq \mathbb{R}^{\kappa - 1}$*, with* $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{X}(\Theta)$*,* $\boldsymbol{\theta}^1, \boldsymbol{\theta}^2 \in \Theta$*. A function* $f : \mathcal{X}(\Theta) \times \Theta \to \mathbb{R}$ *is*

1. ***jointly convex*** *if* f *is convex with respect to the vector* $(\boldsymbol{x}, \boldsymbol{\theta}) \in \mathcal{X}(\Theta) \times \Theta$*; that is,*

$$f(\lambda_1 \boldsymbol{x}^1 + (1 - \lambda_1)\boldsymbol{x}^2, \lambda_1 \boldsymbol{\theta}^1 + (1 - \lambda_1)\boldsymbol{\theta}^2) \leq \lambda_1 f(\boldsymbol{x}^1, \boldsymbol{\theta}^1) + (1 - \lambda_1) f(\boldsymbol{x}^2, \boldsymbol{\theta}^2),$$

2. ***biconvex*** *if* f *is convex with respect to* $\boldsymbol{x}$ *when holding* $\boldsymbol{\theta}$ *constant and if* f *is convex with respect to* $\boldsymbol{\theta}$ *when holding* $\boldsymbol{x}$ *constant; that is,*

$$\begin{aligned} &f(\lambda_1 \boldsymbol{x}^1 + (1 - \lambda_1)\boldsymbol{x}^2, \lambda_2 \boldsymbol{\theta}^1 + (1 - \lambda_2)\boldsymbol{\theta}^2) \leq \\ &\quad \lambda_1 \lambda_2 f(\boldsymbol{x}^1, \boldsymbol{\theta}^1) + (1 - \lambda_1)\lambda_2 f(\boldsymbol{x}^2, \boldsymbol{\theta}^1) + \lambda_1 (1 - \lambda_2) f(\boldsymbol{x}^1, \boldsymbol{\theta}^2) + (1 - \lambda_1)(1 - \lambda_2) f(\boldsymbol{x}^2, \boldsymbol{\theta}^2). \end{aligned}$$

We refer to the right-hand side of the above inequality as the ***biconvex-overestimator*** *of* f*.*

Note that biconvexity reduces to joint convexity if the function f is independent of $\boldsymbol{\theta}$. As a result, in this paper, when we say a function is biconvex, we assume that it is not jointly convex.

Finally, we make use of a version of weak differentiability of a function which is required in Section 3.1.2.3.

pwd **Definition 22.** *A function* $f : \Theta \to \mathbb{R}^n$ *is said to be piecewise differentiable on* Θ *if there exists* $N_f \in \mathbb{N}$ *and a partition* $\{\Theta_i^f\}_{i=1}^{N_f}$ *for* f *such that*

1. *The function* f *is differentiable on each subset* Θ_i^f*,* $i \in \{1, \ldots, N_f\}$ *and*

2. *for all* $\bar{\boldsymbol{t}} \in \Theta_i^f \cap \Theta_j^f$ *and for all* $i, j \in \{1, \ldots, N_f\}, i \neq j$, $\nabla f(\bar{\boldsymbol{t}}) = \lim_{\boldsymbol{t} \in int(\Theta_i^f), \boldsymbol{t} \to \bar{\boldsymbol{t}}} \nabla f(\boldsymbol{t})$ *or* $\nabla f(\bar{\boldsymbol{t}}) = \lim_{\boldsymbol{t} \in int(\Theta_j^f), \boldsymbol{t} \to \bar{\boldsymbol{t}}} \nabla f(\boldsymbol{t})$, *where* ∇f *denotes the gradient of* f *with respect to* $\boldsymbol{t}$, *and* $int(\Theta)$ *denotes the interior of* Θ.

We also include the following theorem from [41], which provides the continuity and piecewise differentiability of $\boldsymbol{x}^*$ and f^*. The theorem is applied at $\boldsymbol{\theta} = \boldsymbol{0}$, but we apply it to $\boldsymbol{\theta} = \hat{\boldsymbol{\theta}}$.

Thm:wkDiff **Theorem 10** ([41])**.** *Let* $\hat{\boldsymbol{\theta}} \in int(\Theta)$. *Suppose the following about program* (3.1).

1. *The functions* f *and* g_i, $i \in \{1, \ldots, m\}$, *are twice continuously differentiable in* $(\boldsymbol{x}, \boldsymbol{\theta})$ *in a neighbourhood of* $(\boldsymbol{x}^*(\hat{\boldsymbol{\theta}}), \hat{\boldsymbol{\theta}})$.

2. *The second-order sufficiency conditions for a local minimum of* (3.1) *hold at* $\boldsymbol{x}^*(\hat{\boldsymbol{\theta}})$, *with associated Lagrange multipliers* $\boldsymbol{u}^*(\hat{\boldsymbol{\theta}})$, *where* $\boldsymbol{u}^* : \Theta \to \mathbb{R}^m$.

3. *The gradients of* g_i, $i \in \{1, \ldots, m\}$, *are linearly independent at* $(\boldsymbol{x}^*(\hat{\boldsymbol{\theta}}), \hat{\boldsymbol{\theta}})$.

4. *The strict complementary slackness condition holds.*

Then, for $\boldsymbol{\theta}$ *in a neighbourhood of* $\hat{\boldsymbol{\theta}}$, *there exist unique, once continuously differentiable vector functions* $\boldsymbol{x}^\dagger : \Theta \to \mathbb{R}^n$, $\boldsymbol{u}^\dagger : \Theta \to \mathbb{R}^m$ *satisfying the second order sufficiency conditions for a local minimum of* (3.1) *such that* $\boldsymbol{x}^\dagger(\hat{\boldsymbol{\theta}}) = \boldsymbol{x}^*(\hat{\boldsymbol{\theta}})$ *and* $\boldsymbol{u}^\dagger(\hat{\boldsymbol{\theta}}) = \boldsymbol{u}^*(\hat{\boldsymbol{\theta}})$, *where* $\boldsymbol{u}^*(\hat{\boldsymbol{\theta}})$ *is the dual solution corresponding to* $\boldsymbol{x}^*(\hat{\boldsymbol{\theta}})$.

Note that, since f and $\boldsymbol{g}$ are biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$, $\boldsymbol{x}^*(\boldsymbol{\theta})$ is not just a local minimum, but a global minimum. Treating Θ as the disjoint union of neighbourhoods, as in Definition 22, we assume that $\boldsymbol{x}^*$ is defined piecewise as the union of $\boldsymbol{x}^\dagger$ over each neighbourhood; hence $\boldsymbol{x}^*$ is piecewise differentiable. Thus we may conclude that f^* is differentiable except at finitely many points, faces, and facets of polytope Θ, those where the solution changes between partitioning subsets $\Theta_i \subset \Theta$.

Observe, once more, that $\boldsymbol{x}^*$, $\bar{\boldsymbol{x}}$, and $\boldsymbol{x}^\dagger$ are vector-valued functions, while $\boldsymbol{x}$ and $\boldsymbol{x}^i$ are vectors. Indeed, for all $\boldsymbol{\theta} \in \Theta$, $\boldsymbol{x}^*(\boldsymbol{\theta}) = \boldsymbol{x}$ for some $\boldsymbol{x} \in \mathbb{R}^n$.

3.0.2 The Approximate Simplex Method [14]

sec:jConASM

Let $\boldsymbol{g} : \mathbb{R}^n \times \Theta \to \mathbb{R}^m$ denote the vector of constraint functions g_i, $i \in \{1, \ldots, m\}$. The authors of [14] apply the following multivariate linear interpolation to solve program (3.1) where f and $\boldsymbol{g}$ are jointly convex. Partition polytope Θ into simplices, and consider each simplex in turn.

After $\boldsymbol{x}^*$ and f^* are linearly interpolated over a given simplex, the error quantifying the difference between $\bar{f}$ and f^* is computed and a point is selected with which to further partition the simplex to refine the error. Without loss of generality, consider simplex $\Theta \subset \mathbb{R}^{\kappa-1}$ and let $\boldsymbol{\theta} \in \Theta$. Considering $\boldsymbol{\vartheta}^i$ to be the vertices of Θ, $i \in \{1, \ldots, \kappa\}$, and $\boldsymbol{x}^i \in \mathbb{R}^n$ the corresponding solution of program (3.1) at $\boldsymbol{\vartheta}^i$, we let

$$M := \begin{bmatrix} 1 & 1 & \ldots & 1 \\ \boldsymbol{\vartheta}^1 & \boldsymbol{\vartheta}^2 & \ldots & \boldsymbol{\vartheta}^\kappa \end{bmatrix} \in \mathbb{R}^{\kappa \times \kappa}, \quad X := \begin{bmatrix} \boldsymbol{x}^1 & \boldsymbol{x}^2 & \ldots & \boldsymbol{x}^\kappa \end{bmatrix} \in \mathbb{R}^{n \times \kappa},$$

$$F := \begin{bmatrix} f(\boldsymbol{x}^1, \boldsymbol{\vartheta}^1) & f(\boldsymbol{x}^2, \boldsymbol{\vartheta}^2) & \ldots & f(\boldsymbol{x}^\kappa, \boldsymbol{\vartheta}^\kappa) \end{bmatrix}^T.$$

Since Θ is a simplex (and thus full-dimensional), M is invertible, and therefore linear interpolations for $\boldsymbol{x}^*(\boldsymbol{\theta})$ and $f^*(\boldsymbol{\theta})$ exist and are given as follows:

$$\bar{\boldsymbol{x}}(\boldsymbol{\theta}) := XM^{-1} \begin{bmatrix} 1 \\ \boldsymbol{\theta} \end{bmatrix}, \quad \bar{f}(\boldsymbol{\theta}) := FM^{-1} \begin{bmatrix} 1 \\ \boldsymbol{\theta} \end{bmatrix}.$$

We extend the ASM proposed in [14] to support f and $\boldsymbol{g}$ being biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$.

3.1 Extension to Biconvexity

sec:biconExt

In extending the results of [14], care must be taken in regards to the feasibility of $\bar{\boldsymbol{x}}$ and the calculation of the error term over each simplex, both of which take advantage of the joint convexity of f and $\boldsymbol{g}$. The feasibility of $\bar{\boldsymbol{x}}$ is ensured in Section 3.1.1, based on the properties of the constraints, $\boldsymbol{g}$, while bounds on the error term are computed in 3.1.2, based on the properties of the objective, f.

3.1.1 Feasibility of the Solution Function

sec:biconFeas

While joint convexity of $\boldsymbol{g}$ guarantees feasibility of $\bar{\boldsymbol{x}}(\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in \Theta$, feasibility must be reconsidered when $\boldsymbol{g}$ is biconvex. We present a constraint qualification, which establishes theoretical feasibility of $\bar{\boldsymbol{x}}(\boldsymbol{\theta})$, and a practical convergence result which is incorporated into the modified ASM in Section 3.2.1.

conQual **Constraint Qualification 1.** *Let $\boldsymbol{g}$ be biconvex, and let $\boldsymbol{\theta} \in \Theta$ and $\boldsymbol{\vartheta}^k$ denote the vertices of polyhedron Θ, $k \in \{1, \dots, \kappa\}$, where κ is the number of vertices of Θ. The approximate solution $\bar{\boldsymbol{x}}(\boldsymbol{\theta})$ is feasible to (3.1) at $\boldsymbol{\theta}$ if, for all $i \in \{1, \dots, m\}$,*

$$\sum_{k \neq \ell = 1}^{\kappa} \lambda_k \lambda_\ell g_i(\boldsymbol{x}^k, \boldsymbol{\vartheta}^\ell) \leq -\sum_{k=1}^{\kappa} {\lambda_k}^2 g_i(\boldsymbol{x}^k, \boldsymbol{\vartheta}^k).$$

Proof. Consider constraint g_i, with $i \in \{1, \dots, m\}$. Let

$$\sum_{k \neq \ell = 1}^{\kappa} \lambda_k \lambda_\ell g_i(\boldsymbol{x}^k, \boldsymbol{\vartheta}^\ell) \leq -\sum_{k=1}^{\kappa} {\lambda_k}^2 g_i(\boldsymbol{x}^k, \boldsymbol{\vartheta}^k).$$

Since $\boldsymbol{\theta} \in \Theta$, there exist $\lambda_k \in [0, 1]$, $k \in \{1, \dots, \kappa\}$, such that $\sum_{k=1}^{\kappa} \lambda_k = 1$, with $\boldsymbol{\theta} = \sum_{k=1}^{\kappa} \lambda_k \boldsymbol{\vartheta}^k$. The approximate solution $\bar{\boldsymbol{x}}(\boldsymbol{\theta})$ is feasible when $g_i(\bar{\boldsymbol{x}}(\boldsymbol{\theta}), \boldsymbol{\theta}) \leq 0$. By construction of $\bar{\boldsymbol{x}}$,

$$\begin{aligned} g_i(\bar{\boldsymbol{x}}(\boldsymbol{\theta}), \boldsymbol{\theta}) &= g_i\left(\sum_{k=1}^{\kappa} \lambda_k \boldsymbol{x}^k, \sum_{k=1}^{\kappa} \lambda_k \boldsymbol{\vartheta}^k\right) \\ &\leq \sum_{k,\ell=1}^{\kappa} \lambda_k \lambda_\ell g_i(\boldsymbol{x}^k, \boldsymbol{\vartheta}^\ell) \\ &= \sum_{k \neq \ell = 1}^{\kappa} \lambda_k \lambda_\ell g_i(\boldsymbol{x}^k, \boldsymbol{\vartheta}^\ell) + \sum_{k=1}^{\kappa} {\lambda_k}^2 g_i(\boldsymbol{x}^k, \boldsymbol{\vartheta}^k), \end{aligned}$$

where the inequality follows because g_i is biconvex. Therefore, $\bar{\boldsymbol{x}}(\boldsymbol{\theta})$ is feasible if the premise holds. □

Intuitively, Constraint Qualification (CQ) 1 means that the approximation of $\boldsymbol{x}^*$ will be feasible as long as the other infeasible solutions are not too infeasible at any given vertex. CQ 1 is utilized to construct a biconvex overestimator bound in Theorem 12. In practice, regardless of whether CQ 1 holds, $\bar{\boldsymbol{x}}$ converges to $\boldsymbol{x}^*$ as Θ is partitioned into subsimplices, as proven below in Proposition 6.

feasProp **Proposition 6.** *Let $\underline{\boldsymbol{\theta}} \in \Theta$, let $\{\Theta_i\}_i$ denote a sequence of simplices such that $\underline{\boldsymbol{\theta}} \in \Theta_i$ for all i, $\Theta_i \subseteq \Theta$ for all i, and $\Theta_i \subset \Theta_{i-1}$ for all $i > 1$, let $\bar{\boldsymbol{x}}^i$ be the linear interpolant of $\boldsymbol{x}^*$ over Θ_i, $\bar{\boldsymbol{x}}^i : \Theta_i \to \mathbb{R}^n$. If the vertices of Θ_i converge to $\underline{\boldsymbol{\theta}}$ as $i \to \infty$, then $\bar{\boldsymbol{x}}^i(\underline{\boldsymbol{\theta}})$ converges to $\boldsymbol{x}^*(\underline{\boldsymbol{\theta}})$.*

Proof. Let $\boldsymbol{\vartheta}^{ij}$ denote the jth vertex of simplex Θ_i; as $i \to \infty$, $\boldsymbol{\vartheta}^{ij}$ converges to $\underline{\boldsymbol{\theta}}$ for all $j \in \{1, \dots, \kappa\}$. By Theorem 10, $\boldsymbol{x}^* : \Theta \to \mathbb{R}^n$ is continuous on Θ_i, so, as $i \to \infty$, $\boldsymbol{x}^*(\boldsymbol{\vartheta}^{ij}) \to \boldsymbol{x}^*(\underline{\boldsymbol{\theta}})$. By construction

of $\bar{\boldsymbol{x}}^i : \mathbb{R}^\kappa \to \mathbb{R}^n$, $\bar{\boldsymbol{x}}^i(\boldsymbol{\vartheta}^{ij}) = \boldsymbol{x}^*(\boldsymbol{\vartheta}^{ij})$ for all $\boldsymbol{\vartheta}^{ij}$ and, for each i, $\bar{\boldsymbol{x}}^i(\underline{\boldsymbol{\theta}})$ is a convex combination of $\boldsymbol{x}^*(\boldsymbol{\vartheta}^{ij})$ for all $j \in \{1, \ldots, \kappa\}$. Hence $\bar{\boldsymbol{x}}^i(\underline{\boldsymbol{\theta}}) \to \boldsymbol{x}^*(\underline{\boldsymbol{\theta}})$ as $i \to \infty$. □

As a result of Proposition 6, the interpolant $\bar{\boldsymbol{x}}$ will approach feasibility as we partition Θ into smaller simplices.

3.1.2 Error Bound Computation

sec:errBd

Consider the furthest distance between the true solution function, f^*, and the linear interpolant, $\bar{f}$, over Θ, that is, consider

$$||f^* - \bar{f}||_\infty = \max_{\boldsymbol{\theta} \in \Theta} \{|\bar{f}(\boldsymbol{\theta}) - f^*(\boldsymbol{\theta})|\}. \tag{3.2}$$ error

The authors of [14] use the joint convexity of f and $\boldsymbol{g}$ to simplify (3.2) to allow for an error calculation of the next point $\boldsymbol{\theta}^{err}$ at which to split Θ into smaller simplices; they then apply the interpolation process to the resulting simplices and continue the process until the desired tolerance is reached. We need to compute $\boldsymbol{\theta}^{err}$ differently when f and $\boldsymbol{g}$ are biconvex. Except for [91], error bounds in relevant interpolation theory involve lower bounds for the error term as demonstrated in [21]. As such we propose two methods of obtaining an upper bound to yield a desired tolerance level—first a method based on the longest edge of the current simplex, followed by a method based on the biconvex overestimator from Definition 21. We compare the two methods in Section 3.2.2.

Let $R \in \mathbb{R}$ and $\boldsymbol{\theta}_c \in \mathbb{R}^{\kappa-1}$ be the radius and center of the hypersphere circumscribing Θ and define $H(f^*) : \Theta \to \mathbb{R}^{\kappa-1} \times \mathbb{R}^{\kappa-1}$ as the Hessian of f^*, made of piecewise second derivatives, with $|H(f^*)|$ being the spectral radius of the same.

3.1.2.1 Longest Edge Method

sec:LEM

Begin by considering linear interpolation errors. When $\kappa = 1$, the error bound on a simplex (interval) $\Theta = [a, b]$ can be computed to be

$$\frac{h^2}{8} \max_{a \le \xi \le b} \left| \frac{d^2 f^*}{d\theta^2}(\xi) \right|,$$

where $h := \max_{1 \le i \le r}(t_i - t_{i-1})$ is the length of the longest subinterval $[t_{i-1}, t_i] \subset [a, b]$; see [10], for example. This bound is extended to $\kappa > 2$ for twice differentiable functions in [94] and simplified in

[91].

waldronThm **Theorem 11.** *Let $f^* : \mathbb{R}^{\kappa-1} \to \mathbb{R}$ be twice differentiable and let $\Theta \subseteq \mathbb{R}^{\kappa-1}$ be a polyhedron with vertices $\boldsymbol{\vartheta}_i$, $i \in \{1, \dots, \kappa\}$. Suppose that $\bar{f}$ is a linear interpolant of f^* using points $\boldsymbol{\vartheta}_i$, $i \in \{1, \dots, \kappa\}$. Then, for each $\boldsymbol{\theta} \in \Theta$, there exists the sharp inequality*

$$|f^*(\boldsymbol{\theta}) - \bar{f}(\boldsymbol{\theta})| \leq \frac{1}{2}(R^2 - ||\boldsymbol{\theta} - \boldsymbol{\theta}_c||_2^2)\left|\left||H(f^*)|\right|\right|_\infty. \tag{3.3}$$ lInterpBd

Further, there exists the uniform bound

$$||f^* - \bar{f}||_\infty \leq \frac{1}{2}(R^2 - \min_{\boldsymbol{\theta} \in \Theta}\{||\boldsymbol{\theta} - \boldsymbol{\theta}_c||_2^2\})\left|\left||H(f^*)|\right|\right|_\infty. \tag{3.4}$$ lInterpUniBd

These bounds correlate to the lengths of the simplex Θ: Theorem 4.1 (v) in [91, p. 84] establishes that (3.4) simplifies to

$$||f^* - \bar{f}||_\infty \leq \frac{\ell^2}{8}\left|\left||H(f^*)|\right|\right|_\infty, \tag{3.5}$$ naiveBd

where ℓ is the length of the longest edge.

Notice that (3.3) and (3.4) in [94] and (3.5) in [91] require twice continuous differentiability of the function being interpolated; however, f^* is only piecewise differentiable over Θ. To account for this, we additionally assume that f^* is piecewise twice differentiable, and that Θ in program (3.1) is partitioned into simplices over which f^* is twice differentiable.

To use (3.5) to partition Θ, assume that $\left|\left||H(f^*)|\right|\right|_\infty$ is bounded by a finite number, M. Therefore interpolating f^* accurately and decreasing the right-hand side of (3.2) requires decrease of ℓ. We then partition Θ in such a way that ℓ^2 is minimized. To achieve an error tolerance of ϵ^{tol}, we require that $\ell_i \leq \sqrt{\frac{8\epsilon^{tol}}{M}}$, where ℓ_i is the length of the longest edge of subsimplex $\Theta_i \subseteq \Theta$. We explore two ways to do this.

Suppose that we split Θ at the center every iteration, as in Figure 3.1a. Note that ℓ_i is constant if we consider Θ_i to always be the leftmost triangle, and Θ_i becomes 'squatter' as we continue splitting. However, this split method can be effective if the derivative of f^* is small along the longest edge of Θ_i, as demonstrated for $\mathbb{R}^2$ in [87].

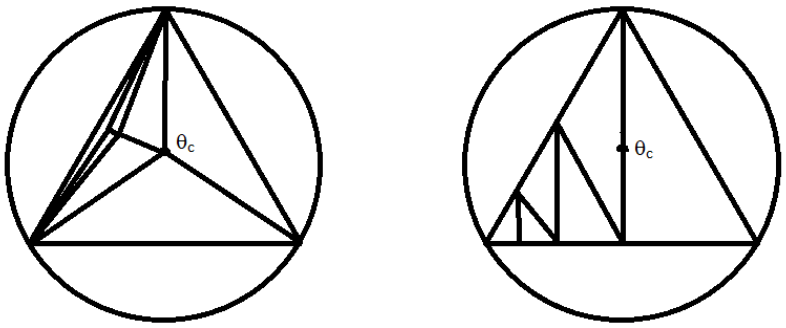


fig:ctrSplit (a) Split simplex at center. fig:sideSplit (b) Split simplex at longest edge.

Figure 3.1: A comparison of two simple methods for partitioning a simplex into smaller simplices; 3.1a does not guarantee error reduction in (3.5), while 3.1b does.

fig:simplexSplits

To guarantee convergence of the right-hand side of (3.5), we choose, instead, to split Θ by putting a hyperplane through the midpoint of the longest edge and all points not incident with said edge. This way, ℓ always decreases, as demonstrated in Figure 3.1b. Indeed, for some $t \in \mathbb{N}$, halving the longest side of the simplex is guaranteed to quarter the error of the interpolation every t iterations. We refer to this method of partitioning Θ as the Longest Edge Method (LEM). Note that, in Figure 3.1, we choose to only split the left side of the triangles to demonstrate the method clearly.

3.1.2.2 Biconvex Overestimator Method

Note that (3.5) fails to utilize the structure of program (3.1). To exploit the properties of program (3.1), through Theorem 12, we prove that $||f^* - \bar{f}||_\infty$ is bounded above when f is biconvex in $(\boldsymbol{x}, \boldsymbol{\theta})$, while, when f is jointly convex in $(\boldsymbol{x}, \boldsymbol{\theta})$, the error bound in [14] applies. We begin with the biconvex case.

Thm:err **Theorem 12.** *Over simplex Θ, the error in* (3.2) *is bounded above as follows:*

$$||f^* - \bar{f}||_\infty \leq \epsilon,$$

where

$$\epsilon := \max\left\{\max_{\boldsymbol{\theta}\in\Theta}\{\bar{f}(\boldsymbol{\theta}) - f^*(\boldsymbol{\theta})\},\ w^{err}\right\}, \tag{3.6}$$ errorBd

and

$$w^{err} := \max_{\lambda\in[0,1]^\kappa, \boldsymbol{\theta}\in\Theta} \left\{ \sum_{i,j=1}^{\kappa} \lambda_i\lambda_j f(\boldsymbol{x}^i, \boldsymbol{\theta}^j) - \bar{f}(\boldsymbol{\theta}) \Big| \boldsymbol{\theta} = \sum_{i=1}^{\kappa} \lambda_i \boldsymbol{\theta}_i \right\}. \tag{3.7}$$ errorWah

Proof. The absolute value in (3.2) yields the following:

$$\max_{\boldsymbol{\theta}\in\Theta}\{|\bar{f}(\boldsymbol{\theta}) - f^*(\boldsymbol{\theta})|\} = \max\left\{ \max_{\boldsymbol{\theta}\in\Theta}\{\bar{f}(\boldsymbol{\theta}) - f^*(\boldsymbol{\theta})\},\ \max_{\boldsymbol{\theta}\in\Theta}\{f^*(\boldsymbol{\theta}) - \bar{f}(\boldsymbol{\theta})\} \right\}.$$

Consider each piece of the maximization separately.

Applying the definition of f^* yields

$$\begin{aligned} \max_{\boldsymbol{\theta}\in\Theta}\{\bar{f}(\boldsymbol{\theta}) - f^*(\boldsymbol{\theta})\} &= \max_{\boldsymbol{\theta}\in\Theta}\{\bar{f}(\boldsymbol{\theta}) - \min_{\boldsymbol{x}\in\mathcal{X}(\Theta)}\{f(\boldsymbol{x},\boldsymbol{\theta})\}\} \\ &= \max_{\boldsymbol{\theta}\in\Theta}\{\bar{f}(\boldsymbol{\theta}) + \max_{\boldsymbol{x}\in\mathcal{X}(\Theta)}\{-f(\boldsymbol{x},\boldsymbol{\theta})\}\} \\ &= \max_{\boldsymbol{x}\in\mathcal{X}(\Theta),\, \boldsymbol{\theta}\in\Theta}\{\bar{f}(\boldsymbol{\theta}) - f(\boldsymbol{x},\boldsymbol{\theta})\}, \end{aligned} \tag{3.8}$$ error1

which is the first part of (3.6).

Now consider $\max_{\boldsymbol{\theta}\in\Theta}\{f^*(\boldsymbol{\theta}) - \bar{f}(\boldsymbol{\theta})\}$: the following reformulation,

$$\begin{aligned} \max_{\boldsymbol{\theta}\in\Theta}\{f^*(\boldsymbol{\theta}) - \bar{f}(\boldsymbol{\theta})\} &= \max_{\boldsymbol{\theta}\in\Theta}\{\min_{\boldsymbol{x}\in\mathcal{X}(\Theta)}\{f(\boldsymbol{x},\boldsymbol{\theta}) - \bar{f}(\boldsymbol{\theta})\}\} \\ &= \max_{\boldsymbol{\theta}\in\Theta}\{-\max_{\boldsymbol{x}\in\mathcal{X}(\Theta)}\{f(\boldsymbol{x},\boldsymbol{\theta})\} - \bar{f}(\boldsymbol{\theta})\}, \end{aligned}$$

does not result in a solvable program, so we construct an overestimator of $f^*(\boldsymbol{\theta})$ using the biconvexity of f.

We follow [66] where it is proven that, if f and $\boldsymbol{g}$ are jointly convex in $(\boldsymbol{x},\boldsymbol{\theta})$, then $f^* : \Theta \to \mathbb{R}$ in (3.1) is convex (and continuous) in $\boldsymbol{\theta}$. Continuity of f^* is guaranteed by [18], but the process from [66] fails to provide convexity of f^* in $\boldsymbol{\theta}$ when f and $\boldsymbol{g}$ are biconvex. However, this process yields an upper bound of (3.2). Consider, first, arbitrary $\boldsymbol{\theta}^1, \boldsymbol{\theta}^2 \in \Theta$ and feasible $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{X}(\Theta)$ such that $f^*(\boldsymbol{\theta}^1) = f(\boldsymbol{x}^1, \boldsymbol{\theta}^1)$ and $f^*(\boldsymbol{\theta}^2) = f(\boldsymbol{x}^2, \boldsymbol{\theta}^2)$. Define $\lambda \in [0,1]$ such that

$$\tilde{\boldsymbol{\theta}} = \lambda\boldsymbol{\theta}^1 + (1-\lambda)\boldsymbol{\theta}^2, \text{ and } \tilde{\boldsymbol{x}} = \lambda\boldsymbol{x}^1 + (1-\lambda)\boldsymbol{x}^2.$$

If CQ 1 holds, then $\tilde{\boldsymbol{x}}$ is feasible, which implies

$$f^*(\tilde{\boldsymbol{\theta}}) = \min_{\boldsymbol{x}\in\mathcal{X}(\tilde{\boldsymbol{\theta}})} f(\boldsymbol{x},\tilde{\boldsymbol{\theta}}) \leq f(\tilde{\boldsymbol{x}},\tilde{\boldsymbol{\theta}})$$

Apply Definition 21 to obtain

$$f(\tilde{\boldsymbol{x}},\tilde{\boldsymbol{\theta}}) \leq \lambda^2 f(\boldsymbol{x}^1,\boldsymbol{\theta}^1) + \lambda(1-\lambda)[f(\boldsymbol{x}^1,\boldsymbol{\theta}^2) + f(\boldsymbol{x}^2,\boldsymbol{\theta}^1)] + (1-\lambda)^2 f(\boldsymbol{x}^2,\boldsymbol{\theta}^2).$$

Extending from using two points $\boldsymbol{\theta}^1$ and $\boldsymbol{\theta}^2$ to using every vertex of Θ, consider $\boldsymbol{\vartheta}^i$, $i \in \{1,\ldots,\kappa\}$, as the ith vertex of simplex Θ. Since $\boldsymbol{x}^i$ is an optimal solution to program (3.1) at $\boldsymbol{\vartheta}^i$, then, for all $\boldsymbol{\theta} \in \Theta$ such that $\boldsymbol{\theta} = \sum_{i=1}^{\kappa} \lambda_i \boldsymbol{\vartheta}^i$, $\lambda_i \geq 0$, $\sum_{i=1}^{\kappa} \lambda_i = 1$, we have

$$f^*(\boldsymbol{\theta}) \leq \sum_{i,j=1}^{\kappa} \lambda_i \lambda_j f(\boldsymbol{x}^i,\boldsymbol{\vartheta}^j), \tag{3.9}$$ biconOverest

where the right-hand side is the biconvex overestimator from Definition 21. Therefore, we can bound the error program by another program as below:

$$\max_{\boldsymbol{\theta}\in\Theta}\{f^*(\boldsymbol{\theta}) - \bar{f}(\boldsymbol{\theta})\} \leq \max_{\lambda,\boldsymbol{\theta}} \left\{ \sum_{i,j=1}^{\kappa} \lambda_i \lambda_j f(\boldsymbol{x}^i,\boldsymbol{\vartheta}^j) - \bar{f}(\boldsymbol{\theta}) \Big| \boldsymbol{\theta} = \sum_{i=1}^{\kappa} \lambda_i \boldsymbol{\vartheta}^i \right\} =: w^{err},$$

which completes the proof. □

To compute ϵ in (3.6), note that $\bar{f}(\boldsymbol{\theta})$ is affine in $\boldsymbol{\theta}$, and $f(\boldsymbol{x},\boldsymbol{\theta})$ is biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$. Therefore, program (3.8) is a biconcave maximization so can be solved by the methods presented in [45], if global optimality is desired, or in [69], if partial optimality is sufficient (for details, see [48]). Similarly, w^{err} in (3.7) can be computed in the following manner.

Since $\bar{f}$ is a linear interpolant, we can write $\bar{f}(\boldsymbol{\theta}) := a^T\boldsymbol{\theta} + b$, so substituting $\boldsymbol{\theta} = \sum_{i=1}^{\kappa} \lambda_i \boldsymbol{\vartheta}^i$ into $\bar{f}(\boldsymbol{\theta})$ yields

$$\bar{f}(\boldsymbol{\theta}) = a^T\boldsymbol{\theta} + b = a^T \sum_{i=1}^{\kappa} \lambda_i \boldsymbol{\vartheta}^i + b = \sum \lambda_i a^T \boldsymbol{\vartheta}^i + b.$$

Then, in (3.7), we really maximize

$$w(\lambda) := \sum_{i,j=1}^{\kappa} \lambda_i \lambda_j f(\boldsymbol{x}^i,\boldsymbol{\vartheta}^j) - \sum \lambda_i a^T \boldsymbol{\vartheta}^i - b. \tag{3.10}$$ wahDef

That is, we solve

$$w^{err} = \max_{\lambda \in [0,1]^\kappa} w(\lambda)$$
$$\text{s.t.} \sum_{i=1}^{\kappa} \lambda_i = 1. \tag{3.11}$$

error2

This is a biconvex program, so can be solved using the methods in [45] or [69]. We take

$$\boldsymbol{\theta}^{err} := \sum_{i=1}^{\kappa} \lambda_i^{err} \boldsymbol{\vartheta}^i,$$

where λ^{err} optimizes (3.7). Most of the time, (3.11) is immediately solvable, as demonstrated now.

Note the solution of program (3.11) occurs at a point where, for all $i \in \{1, \ldots, \kappa\}$,

$$0 = \frac{d\, w}{d\lambda_i} = \sum_{j=1}^{m} \lambda_j [f(\boldsymbol{x}^i, \boldsymbol{\vartheta}^j) + f(\boldsymbol{x}^j, \boldsymbol{\vartheta}^i)] - a^T \boldsymbol{\vartheta}^i,$$
$$1 = \sum_{j=1}^{m} \lambda_j.$$

This system of linear equations can be rewritten to involve a symmetric matrix, K:

$$K \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_\kappa \end{bmatrix} = \begin{bmatrix} a^T \boldsymbol{\vartheta}^1 \\ \vdots \\ a^T \boldsymbol{\vartheta}^\kappa \end{bmatrix},$$

where the ijth element of matrix K is defined as

$$K_{ij} := f(\boldsymbol{x}^i, \boldsymbol{\vartheta}^j) + f(\boldsymbol{x}^j, \boldsymbol{\vartheta}^i).$$

It has been shown that matrices of the form of K are almost surely nonsingular [24]; that is, sampling from a random distribution of matrices of the form K will yield, with probability approaching 1, a nonsingular matrix. As can be seen, if K is nonsingular, then (3.11) is immediately solvable.

An example of how the overestimator, f^*, and interpolant, $\bar{f}$, relate appears in Figure 3.2.

To properly implement the use of (3.6) to compute error bounds on $\bar{f}$ in the Biconvex Overestimator Method (BOM), we show that w^{err} in (3.11) converges to 0. Given this convergence,

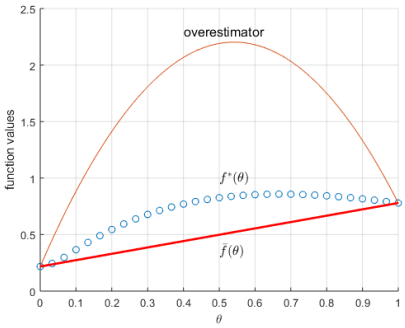


Figure 3.2: The overestimator is on top, $\bar{f}$ is the line on bottom, and the dots denote a discretization of f^* in between.

fig:overest

we use $\boldsymbol{\theta}^{err}$ as a new vertex to partition Θ into smaller simplices.

convProp **Proposition 7.** *Let $\underline{\boldsymbol{\theta}} \in \Theta$. If the vertices of Θ converge to $\underline{\boldsymbol{\theta}}$, then the biconvex overestimator in* (3.9) *of $f^*(\boldsymbol{\theta})$ converges pointwise to $f^*(\boldsymbol{\theta})$.*

Proof. Let $\{\Theta_i\}_i$ denote a sequence of simplices such that $\underline{\boldsymbol{\theta}} \in \Theta_i$ for all i, $\Theta_i \subseteq \Theta$ for all i, and $\Theta_i \subset \Theta_{i-1}$ for all $i > 1$. Let $\boldsymbol{\vartheta}^{ij}$ denote the jth vertex of simplex Θ_i. Note that as $i \to \infty$, $\boldsymbol{\vartheta}^{ij}$ converges to $\underline{\boldsymbol{\theta}}$ for all $j \in \{1, \dots, \kappa\}$. Similarly, let $\boldsymbol{x}^{ij}$ denote an optimal solution to (3.1) at $\boldsymbol{\vartheta}^{ij}$. Recall from [41] that $\boldsymbol{x}^* : \Theta \to \mathbb{R}^n$ is continuous, so, as $i \to \infty$, $\boldsymbol{x}^{ij} = \boldsymbol{x}^*(\boldsymbol{\vartheta}^{ij}) \to \boldsymbol{x}^*(\underline{\boldsymbol{\theta}}) =: \underline{\boldsymbol{x}}$, where $\underline{\boldsymbol{x}}$ is an optimal solution to (3.1) at $\underline{\boldsymbol{\theta}}$. Consequently, because f is biconvex, and biconvex functions are continuous, we have the following, where $\lambda_k \in [0,1]$, $\sum_{k=1}^{\kappa} \lambda_k = 1$:

$$f^*(\underline{\boldsymbol{\theta}}) \leq \sum_{h,k=1}^{\kappa} \lambda_h \lambda_k f(\boldsymbol{x}^{ih}, \boldsymbol{\vartheta}^{ik}) \to \sum_{h,k=1}^{\kappa} \lambda_h \lambda_k f(\underline{\boldsymbol{x}}, \underline{\boldsymbol{\theta}}) = f(\underline{\boldsymbol{x}}, \underline{\boldsymbol{\theta}}) \sum_{h,k=1}^{\kappa} \lambda_k = f(\underline{\boldsymbol{x}}, \underline{\boldsymbol{\theta}}) = f^*(\underline{\boldsymbol{\theta}}).$$

□

Now consider the case where $f : \mathbb{R}^n \times \Theta \to \mathbb{R}$ is independent of $\boldsymbol{\theta}$, that is, f is jointly convex in $(\boldsymbol{x}, \boldsymbol{\theta})$. Then, for all $\boldsymbol{x} \in \mathbb{R}^n$, $f(\boldsymbol{x}, \boldsymbol{\theta}^1) = f(\boldsymbol{x}, \boldsymbol{\theta}^2)$ for all $\boldsymbol{\theta}^1, \boldsymbol{\theta}^2 \in \Theta$, that is, there exists some function $h : \mathbb{R}^n \to \Theta$ such that $f(\boldsymbol{x}, \boldsymbol{\theta}) = h(\boldsymbol{x})$. Then, considering λ defined in (3.7), the following

occurs:

$$\begin{aligned}\sum_{i,j} \lambda_i \lambda_j f(\boldsymbol{x}^i, \boldsymbol{\vartheta}^j) &= \sum_{i,j} \lambda_i \lambda_j h(\boldsymbol{x}^i) \\ &= \sum_i \lambda_i h(\boldsymbol{x}^i) \sum_j \lambda_j \\ &= \sum_i \lambda_i h(\boldsymbol{x}^i).\end{aligned}$$

Consequently, (3.10) becomes

$$w(\lambda) = \sum_i \lambda_i h(\boldsymbol{x}^i) - \sum \lambda_i a^T \boldsymbol{\vartheta}^i - b,$$

so program (3.11) is a linear program when f is jointly convex in $(\boldsymbol{x}, \boldsymbol{\theta})$. Therefore the solution thereof will be a vertex of the standard simplex, that is, $\lambda_i = 1$ for some $i \in \{1, \ldots, \kappa\}$ and $\lambda_j = 0$ for all $j \neq i$.

Following Theorem 12 in this case will then result in choosing $\boldsymbol{\theta}^{err}$ to be a vertex, which does not allow for a full-dimensional partitioning of Θ. However, f is jointly convex in $(\boldsymbol{x}, \boldsymbol{\theta})$; consequently, as in [14], $\bar{f}(\boldsymbol{\theta}) \geq f^*(\boldsymbol{\theta})$, so (3.8) is guaranteed to bound (3.2).

sec:adaptive

3.1.2.3 Improved Convergence of Overestimator Method

The partitioning of Θ based on ϵ in (3.6) is grounded in adaptive mesh theory, which can be found in [71] and [68], and elaborated on in [21]. This theory applies to the estimation of solutions to partial differential equations (PDEs) and, more generally, problems of variation; for more details, see [21], specifically Chapter 9.

Integral to solving PDEs is the Lax-Milgram Theorem, posited here, which guarantees the existence of a unique solution to a variational problem.

Thm:LM **Theorem 13** (Lax-Milgram Theorem [21])**.** *Given a Hilbert space V with corresponding inner-product, a continuous, coercive bilinear form $a : V \times V \to \mathbb{R}$, and a continuous linear functional $F \in V'$, dual to V, there exists a unique $u \in V$ such that $a(u, v) = F(v)$ for all $v \in V$.*

We observe two items of note. First, the dual of a Hilbert space just contains bounded continuous linear functionals defined over said Hilbert Space. Second, in computing error bounds,

[21] uses the L^2-norm; however, we use the L^∞-norm; as such, we present the equivalence of the norms.

Proposition 8. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be bounded and be such that f^2 is integrable. Then the L^2-norm and L^∞-norm are equivalent. That is, there exist constants $c, C \in \mathbb{R}$ such that*

$$c||f||_2 \leq ||f||_\infty \leq C||f||_2.$$

Proof. Note that $||f||_2, ||f||_\infty < \infty$. The result follows by proper scaling with c and C. □

An in-depth look at the equivalence of error bounds across the L^2 and L^∞ norms can be found in Chapter 8 of [21].

In demonstrating the effectiveness of adaptive meshes, the PDEs in [68] and [71] both have upper bounds, but have stronger boundary conditions than [21], which only has a lower bound to the actual error, as [21] provides theory for more general problems of variation fulfilling the more general Theorem 13. The fact that (3.6) is an actual upper bound is essential. Using this upper bound, we only partition simplex Θ into new subsimplices if ϵ from (3.6) is greater than some previously chosen error tolerance. The resulting partition of Θ is the very definition of an adaptive mesh.

To use the Lax-Milgram Theorem, and, in turn, the results on adaptive meshes, we need $f^* : \mathbb{R}^\kappa \to \mathbb{R}$ to be piece-wise differentiable, as stated in [21, p.28–29].

prop:pwd **Proposition 9.** *The optimal value function f^* of program* (3.1) *is piecewise differentiable.*

Proof. Consider $\boldsymbol{\theta} \in \text{int}(\Theta)$. Since $\boldsymbol{x}^*$ is piecewise differentiable by Theorem 10, and f is assumed to be (twice) differentiable, the following holds by application of the generalized chain rule:

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}^*(\boldsymbol{\theta}), \boldsymbol{\theta}) = J_{\boldsymbol{\theta}}(\boldsymbol{x}^*, \boldsymbol{\theta})^T \nabla_{(\boldsymbol{x}^*, \boldsymbol{\theta})} f,$$

where $J_{\boldsymbol{\theta}}(\boldsymbol{x}^*, \boldsymbol{\theta})$ is the Jacobian of $\boldsymbol{x}^*$ and $\boldsymbol{\theta}$ with respect to $\boldsymbol{\theta}$, and $\nabla_{(\boldsymbol{x}^*, \boldsymbol{\theta})} f$ is the vector of partial derivatives of f when treating $\boldsymbol{x}^*$ and $\boldsymbol{\theta}$ as variables, not functions. By definition, $f^*(\boldsymbol{\theta}) = f(\boldsymbol{x}^*(\boldsymbol{\theta}), \boldsymbol{\theta})$ for every $\boldsymbol{\theta} \in \Theta$, so f^* is piecewise differentiable. □

Since f^* is piecewise differentiable, we can apply the Lax-Milgram Theorem 13 to solving program (3.1) in the following manner. Let V be the set of all piecewise differentiable functions fulfilling the boundary conditions for f^*, that is, that $\nabla v(\bar{\boldsymbol{\theta}}) = \nabla f^*(\bar{\boldsymbol{\theta}})$ for all $\bar{\boldsymbol{\theta}} \in \partial\Theta$ where $\partial\Theta$ is

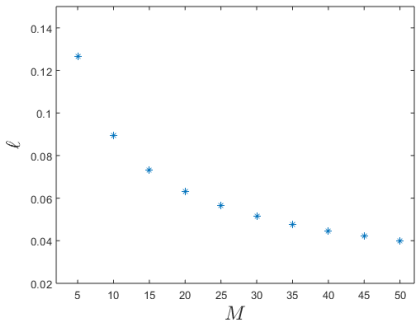


fig:edgeLength

Figure 3.3: Computation of ℓ on the y-axis based on values of M on the x-axis for $\epsilon^{tol} = 0.01$.

the boundary of Θ. That V is a Hilbert space with respect to the standard inner product can be seen through the continuity of derivatives. Then, by letting $a(f^*, v) := \int_\Theta \alpha(\boldsymbol{\theta}) \nabla f^*(\boldsymbol{\theta}) \cdot \nabla v(\boldsymbol{\theta})\, d\boldsymbol{\theta}$, while $F(v) := \int_\Theta \tau(\boldsymbol{\theta}) v(\boldsymbol{\theta})\, d\boldsymbol{\theta}$, the Lax-Milgram Theorem yields that f^* is the solution to an unspecified but existent variational problem of the form

$$\int_\Theta \alpha(\boldsymbol{\theta}) \nabla f^*(\boldsymbol{\theta}) \cdot \nabla v(\boldsymbol{\theta})\, d\boldsymbol{\theta} = \int_\Theta \tau(\boldsymbol{\theta}) v(\boldsymbol{\theta})\, d\boldsymbol{\theta} \text{ for all } v \in V, \tag{3.12}$$

optimPDE

where $\alpha : \Theta \to \mathbb{R}$ is bounded over Θ, and $\tau \in L^2(\Theta)$. Recall that ∇f^* is the gradient of f^*, while $\cdot$ denotes the standard dot product. Note that, were we to need the bounds for problems of variation, then we would need to actually have to specify (3.12).

3.1.2.4 Method Comparison

Using the BOM to select the new vertex $\boldsymbol{\theta}^{err}$ is resource intensive as it requires the solving of two biconvex optimization programs; as a result, we justify its use over the LEM derived from [91]. Notice that, in order to guarantee convergence of $\bar{f}$, the LEM first requires M, an upper bound on the norm of the unknown Hessian of f^*. It next has to break Θ into simplices all with edges shorter than $\sqrt{\frac{8\epsilon^{tol}}{M}}$, where ϵ^{tol} is the acceptable tolerance. Taking $\epsilon^{tol} = 0.01$, Figure 3.3 provides lengths of ℓ based on different bounds M.

To the contrary, the BOM does not require the assumption that f^* be piecewise twice differentiable, nor does it require bounding of the Hessian of f^*. Also, since computing f^* is equivalent to solving a variational problem, adaptive mesh theory can be applied, guaranteeing an improved convergence rate over the LEM. Finally, the BOM adds extra simplices only where the

error bound, ϵ, is outside the tolerance, that is, where $\epsilon > \epsilon^{tol}$; as borne out by the examples, this saves substantially on memory space, which is crucial as the dimension of Θ grows.

3.2 Algorithm and Examples

sec:algEx

Having established two possible termination conditions, we now provide an algorithm for estimating f^*, which we refer to as the multiparametric Biconvex Approximate Simplex Method (BASM). Aside from different error computations, the BASM is analogous to the ASM found in [14]. Given polytope Θ as the parameter space, we partition it into a set of simplices. For each simplex, Θ_t, we solve program (3.1) at each of the simplices and construct interpolants $\bar{f}_t$ and $\bar{\boldsymbol{x}}^t$. We then apply the chosen refinement method (LEM or BOM) and repeat until the desired error tolerance has been reached.

Worth noting is the fact that, although we terminate the algorithm based on properties of $\bar{f}$, the proof of Theorem 12 demonstrates that $f(\bar{\boldsymbol{x}}(\boldsymbol{\theta}), \boldsymbol{\theta})$ will also be rendered accurate by the splitting of Θ at $\boldsymbol{\theta}^{err}$. Also worth recalling is that Proposition 6 demonstrates that $\bar{\boldsymbol{x}}$ approaches feasibility as we split Θ into subsequent simplices. Therefore Algorithm 2 solves program (3.1) to yield approximations to both solution functions and value functions.

3.2.1 Algorithm

sec:alg

To solve program (3.1) when f and $\boldsymbol{g}$ are biconvex, begin in Step 0 of Algorithm 2 by selecting an acceptable error bound $\epsilon^{tol} > 0$ and partitioning polytope Θ into a disjoint set of simplices Θ_t, $t \in \{1, \ldots, s\}$, where s is the number of simplices. Let each Θ_t have vertices $\{\boldsymbol{\vartheta}^{t1}, \ldots, \boldsymbol{\vartheta}^{t\kappa}\}$. At each $t \in \mathbb{N}$, that is, in Step t, compute the solution vectors $\boldsymbol{x}^k(\boldsymbol{\vartheta}^k)$ and objective values $f(\boldsymbol{x}^k(\boldsymbol{\vartheta}^{tk}), \boldsymbol{\vartheta}^{tk})$ for each $k \in \{1, \ldots, \kappa\}$.

If using the LEM, partition Θ_t into two new simplices, which share $\boldsymbol{\theta}^{err}$, the midpoint of the longest edge of Θ_t, as their new vertices. These new subsimplices replace Θ_t in the set of simplices under consideration; repeat until the right-hand side in (3.5) is below ϵ^{tol} for every $t \in \{1, \ldots, s\}$. Recall that the LEM has to assume the existence of some number $M > 0$ bounding the Hessian of f^*.

If using the BOM, compute $\boldsymbol{\theta}^{err}$ as the maximizer of ϵ in (3.6) over Θ_t, and partition Θ_t into κ new simplices having $\boldsymbol{\theta}^{err}$ as a new vertex. As for the LEM, these new simplices replace Θ_t in

the set of simplices. To envision this, consider Θ_1 with vertices $\{\boldsymbol{\vartheta}^1, \boldsymbol{\vartheta}^2, \ldots, \boldsymbol{\vartheta}^\kappa\}$, then there would be new simplices with vertices $\{\boldsymbol{\theta}^{err}, \boldsymbol{\vartheta}^2, \ldots, \boldsymbol{\vartheta}^\kappa\}$, $\{\boldsymbol{\vartheta}^1, \boldsymbol{\theta}^{err}, \ldots, \boldsymbol{\vartheta}^\kappa\}$, $\ldots$, $\{\boldsymbol{\vartheta}^1, \boldsymbol{\vartheta}^2, \ldots, \boldsymbol{\theta}^{err}\}$.

We apply Algorithm 2 to numeric examples using the LEM first, followed by the BOM, and compare the results.

Algorithm 2: Multiparametric Approximate Simplex Algorithm for Biconvex Programs

alg:mp ASM

Input : The initial polytope, Θ, over which to solve (3.1), error tolerance, ϵ^{tol}

Output: piecewise solution function $\bar{\boldsymbol{x}}^K(\boldsymbol{\theta})$ for some terminal $K \in \mathbb{N}$, and piecewise optimal function $\bar{f}^K(\boldsymbol{\theta})$

Step 0: Partition parameter space Θ into a set of simplices Θ_t, $t \in \{1, \ldots, s\}$, and set desirable tolerance and split limit.

alg:step0

Step t: for each $t \in \{1, \ldots, s\}$

alg:stepT

For each vertex $\boldsymbol{\vartheta}^k$ of Θ_t, compute solution vectors $\boldsymbol{x}^k(\boldsymbol{\vartheta}^k)$ and objective values $f(\boldsymbol{x}^k(\boldsymbol{\vartheta}^k), \boldsymbol{\vartheta}^k)$.

Compute linear interpolants $\bar{f}(\boldsymbol{\theta})$ and $\bar{\boldsymbol{x}}(\boldsymbol{\theta})$.

Compute $\boldsymbol{\theta}^{err}$ as midpoint of longest edge of Θ_t,

OR

Compute $\boldsymbol{\theta}^{err}$, maximizer of (3.6),

if *error less than* ϵ^{tol} **then**

| remove t from simplex index set

else

| Split Θ into smaller simplices, with $\boldsymbol{\theta}^{err}$ as new vertex.

| Add new simplex indices to simplex index set.

3.2.2 Examples

sec:ex

All instances are run on a Dell Inspiron 3153 with a 128 GB SSD (6 GB free), 8 GB RAM, running an Intel Core i3-6100U 2.30 GHz processor with two cores and four logical processors running Windows 10 Enterprise. Two MATLAB functions are utilized: *randn* randomly generates the coefficient matrices and vectors using a standard normal distribution, and *fmincon* performs the minimizations at the corresponding steps of Algorithm 2 using the SQP method with the default number of iterations. Multi-Parametric Toolbox (MPT) [52] is utilized to construct the simplices. Prior to continuing, a word must be said on our chosen subroutine: *fmincon* is a black-box algorithm provided by MATLAB to optimize functions, convex or nonconvex, subject to linear and nonlinear constraints, and therefore may return infeasible solutions, with the purpose being to allow the operator to adjust the inputs and try again; as a result, more complicated instances cause *fmincon* to get stuck at infeasible points or spend too much time processing.

To construct the examples, we follow a method similar to that provided in [58], which

was extended from [26]. However, since the generation of a biconvex function is simpler than the generation of a jointly convex function, we only need to add the parameters to the matrices prior to making them positive semi-definite. We choose problem sizes comparable to those chosen in [58]. We let $\Theta = [0.1, 1.1]^{\kappa-1}$, where $\kappa \in \{2, 3, 4\}$. Note that κ is the number of vertices in a simplex, so $\kappa - 1$ is the number of parameters. We choose not to use interval $[0, 1]$ to avoid disappearance of terms at $\boldsymbol{\theta} = \mathbf{0}$; furthermore, we consider the number of variables $n = 5$ for $\kappa \in \{2, 3\}$, and take $n = 3$ for $\kappa = 4$, due to the aforementioned shortcomings of *fmincon*; finally, we consider Hessian bound $M = 30$ for the LEM.

Table 3.1 contains the the time, in seconds, total number of subsimplices, and approximation error, averaged over the number of simplices per instance, as well as the mean, median, and standard deviations thereof, for a set of ten parametric quadratically constrained quadratic programs each with one quadratic constraint and one linear constraint. The parameters in these instances are placed in unrestricted random locations, no more complicated than bilinear or quadratic in $\boldsymbol{\theta}$; that is, $\theta_1\theta_2$ or ${\theta_1}^2$ may be present, but not ${\theta_1}^2\theta_2$. Note that the LEM's error calculation is independent of the program being optimized and therefore has no standard deviation in the number of simplices and error. For all instances, we use an error tolerance of $\epsilon^{tol} = 0.01$, but for three parameters, we limit the number of simplex splits for the LEM and the BOM (twelve for the LEM and ten for the BOM) to keep the number of simplices to a number manageable for the computer, which results in the higher errors in Table 3.1, row $\kappa - 1 = 3$. Notice that, even by giving the LEM the extra two splits, the mean error is four times larger than that of the BOM.

We observe that the LEM maintains consistent results over the instances, and performs faster for one parameter. However, as expected based on the theory presented in Section 3.1.2.3, the BOM outperforms the LEM for two and three parameters; the optimization completes in half the time on average for two and three parameters, storing half as many simplices for two parameters, and a quarter as many for three parameters, achieving a better average error bound in the process. Since the LEM partitions the simplex independently of computations, the standard deviation is significantly lower than that of the BOM: the LEM computes the same number of nonlinear optimizations for every instance, with the only deviation stemming from the random instance and the processor time. When comparing to the BOM, the LEM's computation methods are occasionally detrimental for one parameter, such as instance 3 for $\kappa - 1 = 1$, and detrimental for all subsequent instances aside from instance 7 for $\kappa - 1 = 2$. In contrast, the BOM runs one extra nonlinear

$\kappa - 1$	statistics	LEM			BOM		
		time (sec)	simplices	error	time (sec)	simplices	error
1	instance 1	1.4960	32	0.0037	1.2379	12	0.0031
	instance 2	1.1008	32	0.0037	1.5632	21	0.0054
	instance 3	1.3333	32	0.0037	0.6111	7	0.0046
	instance 4	1.1073	32	0.0037	3.1239	49	0.0045
	instance 5	1.3731	32	0.0037	4.4765	64	0.0032
	instance 6	1.5131	32	0.0037	3.8069	32	0.0071
	instance 7	1.1831	32	0.0037	1.6121	17	0.0059
	instance 8	1.4465	32	0.0037	3.0114	32	0.0054
	instance 9	2.0346	32	0.0037	6.2231	57	0.0054
	instance 10	1.5813	32	0.0037	4.2316	42	0.0053
	mean	1.4169	32	0.0037	2.9898	33.3	0.0050
	median	1.4098	32	0.0037	3.0677	32	0.0054
	std. dev.	0.2757	0	0	1.7497	19.4139	0.0012
2	instance 1	125.7183	2048	0.0073	3.3502	2	0.0085
	instance 2	98.9807	2048	0.0073	0.2563	2	0.0051
	instance 3	130.2507	2048	0.0073	4.2006	26	0.0065
	instance 4	104.9543	2048	0.0073	97.6210	970	0.0051
	instance 5	144.2607	2048	0.0073	29.6928	205	0.0049
	instance 6	151.4086	2048	0.0073	32.9304	227	0.0056
	instance 7	155.2793	2048	0.0073	490.1281	3366	0.0055
	instance 8	143.7425	2048	0.0073	1.5800	12	0.0063
	instance 9	143.7516	2048	0.0073	60.2701	460	0.0054
	instance 10	108.2385	2048	0.0073	2.0482	19	0.0059
	mean	130.6585	2048	0.0073	72.2078	528.9	0.0059
	median	136.9966	2048	0.0073	16.9467	115.5	0.0056
	std. dev.	20.4282	0	0	150.2775	1042.425	0.0011
3	instance 1	1471.2953	24576	0.0438	58.0992	512	0.0059
	instance 2	1605.1566	24576	0.0438	946.8017	7669	0.0098
	instance 3	1540.3899	24576	0.0438	883.0943	8093	0.0105
	instance 4	2023.6621	24576	0.0438	922.1740	6303	0.0076
	instance 5	1212.7538	24576	0.0438	0.3797	6	0.0071
	instance 6	1461.0556	24576	0.0438	53.4763	417	0.0047
	instance 7	1619.6000	24576	0.0438	676.3052	4866	0.0166
	instance 8	2001.1284	24576	0.0438	2547.7292	19291	0.0321
	instance 9	1854.9257	24576	0.0438	1529.4891	11537	0.0142
	instance 10	1402.0011	24576	0.0438	278.0517	2431	0.0073
	mean	1619.2125	24576	0.0438	789.5600	6112.5	0.0116
	median	1572.7733	24576	0.0438	779.6997	5584.5	0.0087
	std. dev.	264.7457	0	0	793.2942	6015.5381	0.0081

Table 3.1: Comparison of LEM and BOM errors, CPU times, and number of simplices with one quadratic constraint, one linear constraint, and $\kappa - 1$ parameters, at ten instances each; averages are included after each set of ten instances. The LEM is applied with a Hessian bound of $M = 30$. Note that, for $\kappa - 1 = 3$, subroutine issues require that the number of variables be dropped from five to three.

tab:exampleErrors

optimization per simplex, and results in a different number of simplices for each instance; depending on the problem being solved, and the subroutine in use, this can greatly influence the time involved, as seen comparing instance 7 to instances 1, 2, 8, and 10 for $\kappa - 1 = 2$.

3.3 Conclusion

We extend the approximate simplex method found in [14] to apply to biconvex objectives and constraints rather than just jointly convex objectives and constraints. We apply said algorithm to a set of random examples using Matlab's *fmincon* as a subroutine, since it has such widespread uses in the engineering community. The termination conditions are twofold: interpolation error bounds from [94] and [91] allow us a choice of new datapoint without error calculations, which we refer to as the LEM; alternatively, we utilize a biconvex overestimator to compute an error estimator, the BOM, and apply properties of adaptive mesh refinement from [21] to justify its use. The LEM has the advantage of having a predetermined number of vertices at which to solve program (3.1), while the BOM enjoys the advantages inherent to adaptive finite element methods as presented in [71], [68], and [21].

Further questions include the comparison of the BOM and LEM with new baseline methods or a hybridization of the two, or the application of different subroutines, as found in the *cvx* and *NLopt* toolboxes; worth investigating, also, is the question of how many changes to a parameter must occur to render parametric programming faster than recomputing the optimal solution each time it is necessary. Currently no computational comparisons exist in the literature, and such comparisons would be useful for the construction of new algorithms. From a more general perspective, the extension of the approximate simplex method from programs containing jointly convex functions to those containing biconvex functions enables consideration of a wider variety of programs, including pooling problems, which tend to be bilinear (and thus biconvex) in nature, as well as various single-objective scalarizations for multi-objective programs.

Chapter 4

A Branch-and-Bound Algorithm for Parametric Mixed-Binary Nonlinear Programs

ch:BnB

[This chapter consists of a paper which will be submitted for review to the *Journal of Global Optimization* under the title of 'A Branch-and-Bound Algorithm for Parametric Mixed-Binary Nonlinear Programs,' coauthored with Dr. Margaret Maria Wiecek.]

4.1 Introduction

intro

Due to inherent rapid shifting in costs and requirements in industry and applied sciences, the desire to compute solutions to mathematical programs for a variety of coefficient values has emerged as a research field over the latter half of the twentieth century. This emergence has given foundation to robust optimization [15] and parametric optimization [42]. Robust optimization considers optimization under uncertainty by considering the worst case in which the uncertainty results. Examples in this area include materials and mechanical engineering problems such as stress-testing car parts or bridge construction: the parts of the car need to be able to endure some maximum force, while the bridge needs to be able to support some maximum weight. In some cases, though, robust optimization is not sufficient: for instance, a regional power grid needs to be able to meet demand

at all times, not just peak times; note that power storage constraints render excess production not practical. Additionally, the pooling problem, a quadratic multicommodity network flow problem modeling flow of materials at uncertain proportions, is of high importance to the petrochemical industry. The properties of this problem naturally imply modeling via parametric mixed binary quadratically constrained quadratic programs; however, the uncertainty of the proportions gives rise to parameters in several locations of the mathematical model, preventing previously devised algorithms from being utilized.

Due to the wealth of precursive literature, we provide a rather brief summary of parametric programming, including the foundational works and the development of the state of the art as it pertains to our own research. For a state of the art in continuous and mixed-integer parametric programming methodology the reader is referred to [85].

One of the earliest considerations of parameters in optimization involves a stochastic view of convex programs [66]. From that point on, properties have been considered purely from a mathematical rather than statistical viewpoint [12, 42, 43] with algorithmic consideration beginning with a penalty algorithm [42]. Since the location of the parameters significantly affects the solution complexity of the resulting parametric programs, the first efforts are focused on models with the parameters in the right-hand-side (RHS) of the constraints. For that parameter location, a theory and algorithm are developed for quadratic programs [29] and later a quadratic approximation algorithm is provided for convex nonlinear programs [28]. A versatile algorithm, to which we refer as the Approximate Simplex Method (ASM), approximates optimal solutions to continuous programs that allow general locations of the parameters and only require that the objective and constraints be jointly convex in the parameters and variables [14]. Parameters make up bilinear terms in the left-hand side of the constraints and more complicated objective functions in [86, 80], while [17] and [23] work with objective and constraint functions polynomial in the parameters by considering Groebner bases to solve the KKT systems. A solution method based on the parametric linear complementarity problem yields closed-form optimal solutions to multiparametric quadratic programs with parameters in general locations in both the objectives and constraints [5]. A barrier method relaxing nonlinear constraints into the objective function allows to compute approximate and exact optimal solutions to different types of multiparametric convex programs [100]. There is also ongoing consideration of the interplay between first-order partial differential equations and the KKT conditions, since the derivatives therein are functions of parameters [9, 86].

At the same time, progress has also been made with regards to mixed integer parametric programming starting with linear programs and RHS parameters [2, 3, 32, 76] and continuing into linear programs with parameters in general locations [64, 70, 75, 99]. In particular, in [75, 99], the parameter in the LHS of the constraints is considered as part of bilinear terms in the parameter and variable that are treated with the McCormick relaxation. Algorithms for mixed-integer variables and RHS-parameters in quadratic programs [29, 11] and convex nonlinear programs [28, 31] are also developed. The methodology for mixed-integer parametric programming involves decomposition methods leading to cutting plane algorithms [31, 29], quadratic approximation [28], and branch-and-bound (BB) methods. The latter literature forms two broad divisions: the static, fixed-tree enumeration group selects integer variables based solely on the construction of the tree [11, 32], while the dynamic enumeration group makes deliberate choices of which relaxed variables to set integer [2, 3, 64, 70, 75, 99].

Additionally, in the multicriterion decision-making (MCDM) realm, groundbreaking research is being done to introduce parameters into multiobjective programs, as seen in [58]. This investigation has in turn led to an extension of the ASM [14], which is developed in [77], and called the Biconvex Approximate Simplex Method (BASM).

Note that the literature, by-and-large, refers to parametric programs with functions linear or quadratic in the variables, and functions bilinear in the variables and parameters began to be introduced only in the last decade. We consider parametric mixed binary nonlinear programs involving parameters and binary variables in general locations throughout the program, as inspired by pooling problems. Due to the complications induced by parameters in general locations, this type of program is currently unsolved. However, based on recent work [77], we are now able to develop a static enumeration BB algorithm using the BASM [77] as a solver at each node. Additionally, we provide some rudimentary benchmarking for the algorithm applied to random examples, as in [58, 77], which we hope will benefit future research by providing a ready template against which to compare newer algorithms.

We begin by formulating the problem and defining the terms we need in Section 4.2. In Section 4.3 we consider possible subroutines with which to solve the node problems and demonstrate the superiority of the BASM. We then proceed with a statement of the complete BB algorithm, followed by a consideration of illustrative examples along with randomly generated instances in Section 4.4. We close the paper in Section 4.5 with concluding remarks and further research directions.

sec:BBdefn

4.2 Problem Statement and Definitions

Let $\kappa \geq 2$, $\Theta \in \mathbb{R}^{\kappa-1}$ be a full-dimensional polytope of parameters. In practice, Θ will be first reduced into a union of simplices of vertex number κ. Let $f, g_i : \mathbb{R}^n \times \{0,1\}^m \times \Theta \to \mathbb{R}$, $i \in \{1, \ldots, nCon\}$ be convex and twice differentiable in $(\boldsymbol{x}, \mathbf{y}) \in \mathbb{R}^{n+m}$ and convex in $\boldsymbol{\theta}$, with $\boldsymbol{\theta} \in \Theta$ in general locations. Consider the following parametric mixed-binary nonlinear program.

$$
\begin{aligned}
f^*(\boldsymbol{\theta}) := \min_{\boldsymbol{x} \in \mathbb{R}^n_{\geqq},\, \mathbf{y} \in \{0,1\}^m} \; & f(\boldsymbol{x}, \mathbf{y}, \boldsymbol{\theta}) \\
\text{s.t. } & g_i(\boldsymbol{x}, \mathbf{y}, \boldsymbol{\theta}) \leq 0,\; i \in \{1, \ldots, nCons\} \qquad \text{(01BNLP)} \quad \texttt{m01nlp} \\
& A\boldsymbol{x} + B\mathbf{y} + E\boldsymbol{\theta} + \boldsymbol{d} \leq \mathbf{0} \\
& \boldsymbol{\theta} \in \Theta.
\end{aligned}
$$

Let $w \in \{0, 1, \ldots, m\}$, $\mathbf{y}^\dagger \in \{0,1\}^{m-w}$ be fixed integers, relax $\mathbf{y} \in [0,1]^w$, and $\tilde{\mathbf{y}} = \begin{bmatrix} \mathbf{y}^\dagger \\ \mathbf{y} \end{bmatrix}$. For the sake of the following definitions, let $\tilde{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} \in \mathbb{R}^n_{\geqq} \times [0,1]^w$ be all the continuous variables in the relaxed program. Let the feasible set be defined as

$$
\mathcal{X}(\boldsymbol{\theta}) := \{\tilde{\boldsymbol{x}} \in \mathbb{R}^n_{\geqq} \times [0,1]^w : g_i(\boldsymbol{x}, \tilde{\mathbf{y}}, \boldsymbol{\theta}) \leq 0,\; i = 1, \ldots, nCons,\; A\boldsymbol{x} + B\tilde{\mathbf{y}} + E\boldsymbol{\theta} + \boldsymbol{d} \leq \mathbf{0}\},
$$

where $\boldsymbol{\theta} \in \Theta$, $\mathbb{R}^n_{\geqq} := \{\boldsymbol{t} \in \mathbb{R}^n : t_i \geq 0,\; i = 1, \ldots, n\}$, and $\mathcal{X}(\Theta) := \cup_{\boldsymbol{\theta} \in \Theta} \mathcal{X}(\boldsymbol{\theta})$. Then we have the following continuous relaxed program.

$$
\tilde{f}(\boldsymbol{\theta}) := \min_{\tilde{\boldsymbol{x}} \in \mathcal{X}(\boldsymbol{\theta})} f(\boldsymbol{x}, \tilde{\mathbf{y}}, \boldsymbol{\theta}), \qquad \text{(node)} \quad \texttt{node}
$$

where $\boldsymbol{\theta} \in \Theta$. For simplicity, assume that (node) is completely solvable (with nonempty feasible region) for all $\theta \in \Theta$; if not, then compute the feasibility region for that node and reset Θ accordingly. If $w = 0$, then (node) is referred to as a leaf node; otherwise, it is referred to as a branch node.

BBsolnDef **Definition 23.** *1. The **optimal solution function** to* (node)*, $\tilde{\boldsymbol{x}}^* : \Theta \to \mathbb{R}^n$, is defined by*

$$\tilde{\boldsymbol{x}}^* : \boldsymbol{\theta} \mapsto argmin_{\tilde{\boldsymbol{x}} \in \mathcal{X}(\boldsymbol{\theta})} f(\boldsymbol{x}, \boldsymbol{\theta}), \quad \tilde{\boldsymbol{x}}^* := \begin{bmatrix} \boldsymbol{x}^* \\ \boldsymbol{y}^* \end{bmatrix}, \tilde{\boldsymbol{y}}^* := \begin{bmatrix} \boldsymbol{y}^\dagger \\ \boldsymbol{y}^* \end{bmatrix},$$

*and the **optimal value function**, $f^*(\boldsymbol{\theta}) : \Theta \to \mathbb{R}$, is defined by*

$$f^* : \boldsymbol{\theta} \mapsto f(\boldsymbol{x}^*(\boldsymbol{\theta}), \tilde{\boldsymbol{y}}^*(\boldsymbol{\theta}), \boldsymbol{\theta})$$

where $\boldsymbol{\theta} \in \Theta$.

*2. If $\boldsymbol{\vartheta}^i$ is a vertex of Θ, $i \in \{1, \ldots, \kappa\}$, then, for all i, the **vertex solution** $\tilde{\boldsymbol{x}}^i := \tilde{\boldsymbol{x}}^*(\boldsymbol{\vartheta}^i)$ is an optimal solution of program* (node) *at $\boldsymbol{\vartheta}^i$; that is, $f^*(\boldsymbol{\vartheta}^i) = f(\tilde{\boldsymbol{x}}^i, \boldsymbol{\vartheta}^i)$.*

*3. Define $\bar{\boldsymbol{x}} : \Theta \to \mathbb{R}^n$ and $\bar{f} : \Theta \to \mathbb{R}$ to be the **linear interpolants** of $\tilde{\boldsymbol{x}}^*$ and f^* respectively. That is, if $\Theta \subset \mathbb{R}^{\kappa-1}$ is a simplex with vertices $\boldsymbol{\vartheta}^i$, and $i \in \{1, \ldots, \kappa\}$,*

$$M := \begin{bmatrix} 1 & 1 & \ldots & 1 \\ \boldsymbol{\vartheta}^1 & \boldsymbol{\vartheta}^2 & \ldots & \boldsymbol{\vartheta}^\kappa \end{bmatrix} \in \mathbb{R}^{\kappa \times \kappa}, \quad X := \begin{bmatrix} \boldsymbol{x}^1 & \boldsymbol{x}^2 & \ldots & \boldsymbol{x}^\kappa \end{bmatrix} \in \mathbb{R}^{n \times \kappa},$$

$$F := \begin{bmatrix} f(\boldsymbol{x}^1, \boldsymbol{\vartheta}^1) & f(\boldsymbol{x}^2, \boldsymbol{\vartheta}^2) & \ldots & f(\boldsymbol{x}^\kappa, \boldsymbol{\vartheta}^\kappa) \end{bmatrix}^T$$

then

$$\bar{\boldsymbol{x}}(\boldsymbol{\theta}) := XM^{-1} \begin{bmatrix} 1 \\ \boldsymbol{\theta} \end{bmatrix}, \quad \bar{f}(\boldsymbol{\theta}) := FM^{-1} \begin{bmatrix} 1 \\ \boldsymbol{\theta} \end{bmatrix}, \text{ for all } \boldsymbol{\theta} \in \Theta.$$

When Θ is a polytope, the linear interpolants are the piecewise union of the interpolants over the simplices making up Θ.

Two generalizations of the convexity of a function are defined below.

BBbiconDef **Definition 24.** *Let $\lambda_1, \lambda_2 \in [0, 1]$; consider convex spaces $\mathcal{X}(\Theta) \subseteq \mathbb{R}^n$ and $\Theta \subseteq \mathbb{R}^{\kappa-1}$, with $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{X}(\Theta)$, $\boldsymbol{\theta}^1, \boldsymbol{\theta}^2 \in \Theta$. A function $f : \mathcal{X}(\Theta) \times \Theta \to \mathbb{R}$ is*

*1. **jointly convex** if f is convex with respect to the vector $(\boldsymbol{x}, \boldsymbol{\theta}) \in \mathcal{X}(\Theta) \times \Theta$; that is,*

$$f(\lambda_1 \boldsymbol{x}^1 + (1 - \lambda_1)\boldsymbol{x}^2, \lambda_1 \boldsymbol{\theta}^1 + (1 - \lambda_1)\boldsymbol{\theta}^2) \leq \lambda_1 f(\boldsymbol{x}^1, \boldsymbol{\theta}^1) + (1 - \lambda_1) f(\boldsymbol{x}^2, \boldsymbol{\theta}^2),$$

2. ***biconvex*** *if f is convex with respect to $\boldsymbol{x}$ when holding $\boldsymbol{\theta}$ constant and if f is convex with respect to $\boldsymbol{\theta}$ when holding $\boldsymbol{x}$ constant; that is,*

$$f(\lambda_1 \boldsymbol{x}^1 + (1-\lambda_1)\boldsymbol{x}^2, \lambda_2 \boldsymbol{\theta}^1 + (1-\lambda_2)\boldsymbol{\theta}^2) \leq$$
$$\lambda_1\lambda_2 f(\boldsymbol{x}^1, \boldsymbol{\theta}^1) + (1-\lambda_1)\lambda_2 f(\boldsymbol{x}^2, \boldsymbol{\theta}^1) + \lambda_1(1-\lambda_2) f(\boldsymbol{x}^1, \boldsymbol{\theta}^2) + (1-\lambda_1)(1-\lambda_2) f(\boldsymbol{x}^2, \boldsymbol{\theta}^2).$$

We refer to the right-hand side of the above inequality as the ***biconvex-overestimator*** *of f.*

Note that f and g are assumed to be jointly convex in $\boldsymbol{x}$ and $\mathbf{y}$, and biconvex in $(\boldsymbol{x}, \mathbf{y})$ and $\boldsymbol{\theta}$.

4.3 Potential Subroutines

sec:sub

We provide a brief summary of each of the three possible subroutines with which we could solve the node problems.

Let $\mathbf{y}^\dagger \in \{0,1\}^{m-w}$ be fixed; define $f : \mathbb{R}^{n+w} \times \Theta \to \mathbb{R}$, $\boldsymbol{g} : \mathbb{R}^{n+w} \times \Theta \to \mathbb{R}^{nCon}$ as for (node). That is, f and g_i are biconvex in $\boldsymbol{x} \in \mathbb{R}^{n+w}$ and $\boldsymbol{\theta} \in \Theta$. Consider the following general multiparametric nonlinear program, which is a special case of (node), but with $\boldsymbol{x} = \tilde{\boldsymbol{x}}$ for simplicity:

$$\min_{\boldsymbol{x} \in \mathcal{X}(\boldsymbol{\theta})} f(\boldsymbol{x}; \boldsymbol{\theta}) \tag{4.1}$$

sec3prog

where $\boldsymbol{\theta} \in \Theta$ and

$$\mathcal{X}(\boldsymbol{\theta}) := \{\boldsymbol{x} \in \mathbb{R}^n_{\leqq} \times [0,1]^w : \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{\theta}) \leq \mathbf{0},\ A\boldsymbol{x} + E\boldsymbol{\theta} + \boldsymbol{d} \leq \mathbf{0}\}.$$

We begin with an extension of the quadratic approximation provided by [29], followed by a restatement of the BASM constructed in [77] and a consideration of a dual ascent algorithm found in [63].

4.3.1 Generalized Quadratic Approximation

sec:bConQAExt

When parameters are only in the right-hand side of the constraints, an approximation method which reduces the optimization program to a quadratic program can be utilized, as demon-

strated in [29]. An extension of this method to parameters in broader locations to obtain a similarly analytic solution is worth exploring. We designate this extension the multiparametric General Quadratic Approximation (mpGQA). Using Taylor polynomial expansions, approximate f via a second-order approximation, and $\boldsymbol{g}$ via a first-order approximation at $\boldsymbol{x}^*$ found at the global minimum $(\boldsymbol{x}^*, \boldsymbol{\theta}^*)$ to construct the following quadratic program out of (4.1)

$$\min_{\boldsymbol{x} \in \mathbb{R}^{n+w}_{\geqq}} \frac{1}{2}\boldsymbol{x}^T Q(\boldsymbol{\theta})\boldsymbol{x} + p^T(\boldsymbol{\theta})\boldsymbol{x} + c(\boldsymbol{\theta})$$

$$\text{s.t. } W(\boldsymbol{\theta})\boldsymbol{x} \leq b(\boldsymbol{\theta}) \tag{4.2}$$ quadApprox

where $\boldsymbol{\theta} \in \Theta$. Note (4.3), (4.4), and (4.5) are from the second-order approximation, while (4.6) and (4.7) are from the first-order approximation.

$$Q(\boldsymbol{\theta}) := \nabla^2_{\boldsymbol{x}} f(\boldsymbol{x}^*; \boldsymbol{\theta}), \tag{4.3}$$ SO1

$$p(\boldsymbol{\theta}) := \nabla^T_{\boldsymbol{x}} f(\boldsymbol{x}^*; \boldsymbol{\theta}) - \nabla^2_{\boldsymbol{x}} f(\boldsymbol{x}^*; \boldsymbol{\theta})\boldsymbol{x}^*, \tag{4.4}$$ SO2

$$c(\boldsymbol{\theta}) := f(\boldsymbol{x}^*; \boldsymbol{\theta}) - \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^*; \boldsymbol{\theta})\boldsymbol{x}^* + \frac{1}{2}\boldsymbol{x}^{*T}\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x}^*; \boldsymbol{\theta})\boldsymbol{x}^*; \tag{4.5}$$ SO3

$$W(\boldsymbol{\theta}) := \nabla_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x}^*; \boldsymbol{\theta}) \tag{4.6}$$ FO1

$$b(\boldsymbol{\theta}) := \nabla_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x}^*; \boldsymbol{\theta})\boldsymbol{x}^* - g(\boldsymbol{x}^*; \boldsymbol{\theta}). \tag{4.7}$$ FO2

Note that $Q(\boldsymbol{\theta})$ is positive semi-definite for all $\boldsymbol{\theta}$ by the assumption that f is convex in $\boldsymbol{x}$ for all $\boldsymbol{\theta}$. By applying the Karush-Kuhn-Tucker conditions with $\boldsymbol{u} : \Theta \to \mathbb{R}^m$ being the dual variable, and overloading notation so that now $A(\boldsymbol{\theta})$ contains only the active constraints, we get

$$Q(\boldsymbol{\theta})\boldsymbol{x} + p^T(\boldsymbol{\theta}) + W^T(\boldsymbol{\theta})\boldsymbol{u} = \boldsymbol{0} \tag{4.8}$$ kkt1

$$W(\boldsymbol{\theta})\boldsymbol{x} = b(\boldsymbol{\theta}) \tag{4.9}$$ kkt2

From (4.8), if $Q(\boldsymbol{\theta})$ is positive definite (and thus invertible) we obtain

$$\boldsymbol{x}(\boldsymbol{\theta}) = -Q^{-1}(\boldsymbol{\theta})(p^T(\boldsymbol{\theta}) + W^T(\boldsymbol{\theta})\boldsymbol{u}(\boldsymbol{\theta}))$$

and substituting into (4.9) results in

$$\boldsymbol{u}(\boldsymbol{\theta}) = -(W(\boldsymbol{\theta})Q^{-1}(\boldsymbol{\theta})W^T(\boldsymbol{\theta}))^{-1}(b(\boldsymbol{\theta}) + W(\boldsymbol{\theta})Q^{-1}(\boldsymbol{\theta})p^T(\boldsymbol{\theta})).$$

Note that $\boldsymbol{x}(\boldsymbol{\theta})$ in (4.8) is optimal for program (4.2). Note that, for (4.2), Q^{-1} is a function of $\boldsymbol{\theta}$, unlike in [29].

In general, the accuracy of each approximation is considered to determine whether Θ needs to be partitioned into regions on which new approximations must be made, and if so, where those partitions occur. The process is then repeated on each partition until a sufficient approximation has been obtained. As seen in [62, 63], error bounds on (4.2) can be computed to determine whether Θ must be further partitioned to allow for calculations of subsequent new global minima so as to obtain sufficient approximations to program (4.1). These partitions, which occur both here and in the following section for the BASM, are referred to as invariance or critical regions (CRs); we opt for the latter.

4.3.2 Biconvex Approximate Simplex Method

sec:BASM

Consider program (4.1). Rather than solving a quadratic program comprised of quadratic approximations of the objective and linear approximations of the constraints, we can instead form a linear interpolation of the optimal value function and solution function. Designed in [14], this interpolation method is utilized successfully in [58]. In [77], it is extended to biconvex problems and termed the multiparametric Biconvex Simplex Method (BASM). Since (node) is a biconvex multiparametric program, the BASM is applicable. The process is the same as detailed below, but the error calculation found in [14] has been adjusted to account for biconvexity. Since Θ is assumed to be a polytope, we can partition it into simplices sharing only faces; therefore, assume that Θ is a simplex with κ vertices, $\boldsymbol{\vartheta}^i$, $i \in \{1, \dots, \kappa\}$. Let $\boldsymbol{x}^i \in \mathbb{R}^{n+w}$ be the corresponding solution of program (4.1) at $\boldsymbol{\vartheta}^i$. Form the following matrices to construct a linear interpolation over Θ:

$$M := \begin{bmatrix} 1 & 1 & \dots & 1 \\ \boldsymbol{\vartheta}^1 & \boldsymbol{\vartheta}^2 & \dots & \boldsymbol{\vartheta}^\kappa \end{bmatrix} \in \mathbb{R}^{\kappa \times \kappa}, \quad X := \begin{bmatrix} \boldsymbol{x}^1 & \boldsymbol{x}^2 & \dots & \boldsymbol{x}^\kappa \end{bmatrix} \in \mathbb{R}^{(n+w)\times\kappa},$$

$$F := \begin{bmatrix} f(\boldsymbol{x}^1, \boldsymbol{\vartheta}^1) & f(\boldsymbol{x}^2, \boldsymbol{\vartheta}^2) & \dots & f(\boldsymbol{x}^\kappa, \boldsymbol{\vartheta}^\kappa) \end{bmatrix}^T.$$

Note that Θ is a simplex and thus full-dimensional, so M is invertible. Let the linear interpolants for the optimal node solution function and optimal node value function be $\boldsymbol{x}^*(\boldsymbol{\theta})$ and $f^*(\boldsymbol{\theta})$, respectively:

$$\bar{\boldsymbol{x}}(\boldsymbol{\theta}) := XM^{-1} \begin{bmatrix} 1 \\ \boldsymbol{\theta} \end{bmatrix}, \quad \bar{f}(\boldsymbol{\theta}) := FM^{-1} \begin{bmatrix} 1 \\ \boldsymbol{\theta} \end{bmatrix}.$$

In general, the BASM returns a partitioning of a polytope Θ into a set of simplex CRs with corresponding continuous piecewise linear functions $\boldsymbol{x}^*$ and f^*. To determine this partitioning, an error bound is computed by maximizing the distance between the biconvex overestimator of f, analogous to the convex overestimator, and the linear interpolant $\bar{f}$. This optimization treats θ as a variable to obtain the location at which to further partition. There is not currently an upper bound on error for adaptive mesh methods for problems of this form, so the very existence of this bound makes it valuable.

sec:mpSA

4.3.3 Subgradient Method

Alternatively, as proposed in [7] and implemented in [62, 63], a multiparametric dual gradient ascent can be considered as well. Let $\bar{\mathbb{C}}(\Theta, \mathbb{R}^{nCon})$ be the set of bounded piecewise continuous functions mapping Θ to $\mathbb{R}^{nCon}$. Assume that $\boldsymbol{x} \in \bar{\mathbb{C}}(\Theta, \mathbb{R}^{n+w}_{\geqq})$, and use Lagrange multiplier $\boldsymbol{u} \in \bar{\mathbb{C}}(\Theta, \mathbb{R}^{nCon}_{\geqq})$. to obtain the Lagrangian function associated with (4.1), $\tilde{\mathcal{L}} : \mathbb{R}^{n+m} \times \Theta \to \mathbb{R}$, defined by

$$\tilde{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u}; \boldsymbol{\theta}) := f(\boldsymbol{x}; \boldsymbol{\theta}) + \sum_{i=1}^{nCon} u_i(\theta) g_i(\boldsymbol{x}; \boldsymbol{\theta}) \tag{4.10}$$ dualFunc

Then the Lagrangian relaxation of (4.1) is

$$\begin{aligned} \tilde{\phi}(\boldsymbol{u}; \boldsymbol{\theta}) = \min_{\boldsymbol{x}} \ & \mathcal{L}(\boldsymbol{x}, \boldsymbol{u}; \boldsymbol{\theta}) \\ \text{s.t. } & A\boldsymbol{x} + E\boldsymbol{\theta} + \boldsymbol{d} \leq 0 \\ & \boldsymbol{x} \in \mathbb{R}^{n+w}_{\geqq}, \end{aligned} \tag{4.11}$$ lagRel

where $\boldsymbol{\theta} \in \Theta$.

By parametric strong duality [63], we can maximize $\tilde{\phi}(\boldsymbol{u};\boldsymbol{\theta})$ with respect to $\boldsymbol{u}$ using a subgradient ascent for an optimization involving only nonnegativity constraints; the subgradients in question being $g_i(\boldsymbol{x};\boldsymbol{\theta})$. By alternating solving (4.11) for $\boldsymbol{x}$ and stepping $\boldsymbol{u}$ we can obtain the optimal solution for (4.1) The alternating between gradient steps and re-solving of (4.11) continues until a desired error tolerance is achieved (for more details, see [62, 63]).

To optimize program (4.11), we can use either the mpGQA or the BASM methods, both of which require $\tilde{\mathcal{L}}$ to be biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$. Since f is already assumed to be biconvex, we specifically need $u_i(\boldsymbol{\theta})g_i(\boldsymbol{x},\boldsymbol{\theta})$ to be biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$ for all $i \in \{1,\ldots,nCon\}$. In this approach, we choose to estimate the Lagrange multipliers to be affine functions of θ. That is,

$$\bar{u}_i(\boldsymbol{\theta}) = \boldsymbol{\alpha}_i^T\boldsymbol{\theta} + \beta_i, \quad i \in \{1,\ldots,nCon\} \tag{4.12}$$ affLM

where $\boldsymbol{\alpha}_i \in \mathbb{R}^\kappa$, $\beta_i \in \mathbb{R}$. Note that the true optimal dual solutions are nonlinear functions of $\boldsymbol{\theta}$, but are approximated as affine functions of $\boldsymbol{\theta}$. Therefore we introduce the additional assumption that g_i are affine in $\boldsymbol{\theta}$ and convex in $\boldsymbol{x}$ for all $i \in \{1,\ldots,nCon\}$. Then we can rewrite g_i as follows:

$$g_i(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{\gamma}_i(\boldsymbol{x})^T\boldsymbol{\theta} + \delta_i(\boldsymbol{x})$$

where $\boldsymbol{\gamma}_i : \mathbb{R}^{n+w} \to \mathbb{R}^\kappa$, $\delta_i : \mathbb{R}^{n+w} \to \mathbb{R}$ are convex functions of $\boldsymbol{x}$. Thus, for each i,

$$\begin{aligned}\bar{u}_i(\boldsymbol{\theta})g_i(\boldsymbol{x},\boldsymbol{\theta}) &= (\boldsymbol{\alpha}_i^T\boldsymbol{\theta} + \beta_i)(\boldsymbol{\gamma}_i(\boldsymbol{x})^T\boldsymbol{\theta} + \delta_i(\boldsymbol{x})) \\ &= \boldsymbol{\theta}^T\boldsymbol{\alpha}_i\boldsymbol{\gamma}_i(\boldsymbol{x})^T\boldsymbol{\theta} + (\beta_i\boldsymbol{\gamma}_i(\boldsymbol{x}) + \delta_i(\boldsymbol{x})\boldsymbol{\alpha}_i)^T\theta + \beta_i\boldsymbol{\delta}_i(\boldsymbol{x}).\end{aligned}$$

Consequently, for $\tilde{\mathcal{L}}$ to be biconvex in $\boldsymbol{x}$ and $\boldsymbol{\theta}$, the 1-rank matrix $\boldsymbol{\alpha}_i\boldsymbol{\gamma}_i(\boldsymbol{x})^T$ must be positive semi-definite, which is not straightforward to guarantee.

4.3.4 Subroutine Comparison

subroutineCompare

We have discussed three possible subroutines to use in a BB algorithm: the mpGQA, the mpSA and the BASM. While the mpGQA can theoretically enable solving of programs, the inversion of a matrix containing parameters frequently results in complicated (i.e., nonpolynomial) solutions.

As a result, we choose not to pursue application of the mpGQA for parameters in general locations as a subroutine.

Before we choose between the BASM and the mpSA, we make an observation about the behavior of the latter. We recognize a difference between a general dual ascent algorithm and a multiparametric dual ascent algorithm. In both algorithms, the update of the dual optimization variable is made with a step size λ_k in iteration k. Since in the nonparametric case, the dual variable is a point-value vector, and in the parametric case, it is a function-valued vector, using a λ_k independent of $\boldsymbol{\theta}$ can introduce failure to converge in a timely manner, depending on the nature of true dual solution $\boldsymbol{u}^* : \Theta \to \mathbb{R}^{nCon}$. We present a simple example, constructed specifically to outline this shortcoming.

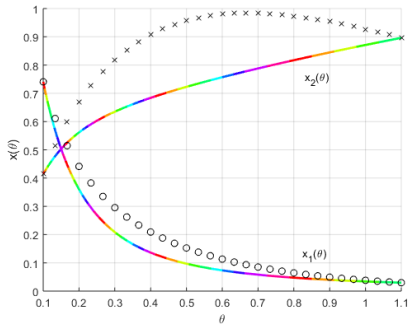
Consider the following single parameter convex nonlinear program:

$$
\begin{aligned}
f^*(x) = \min_{x_1, x_2 \in \mathbb{R}_{\geq}} \; & f(x) = 400\theta^2 {x_1}^2 + 9{x_2}^2 \\
\text{s.t. } & g_1(x,\theta) = (x_1 - 2)^2 + (x_2 - 2)^2 - 4 - \theta \leq 0 \\
& g_2(x,\theta) = 2x_1 + x_2 - \theta - 2 \leq 0 \\
& x_1, x_2 \geq 0, \; \theta \in [0.1, 1.1].
\end{aligned}
\tag{4.13}
$$

breakSA

Let $\boldsymbol{x} \in \mathbb{R}^2_{\geq}$, $\theta \in [0.1, 1.1]$; θ has been chosen to not contain 0 so as to prevent disappearance of the x_1 term in the objective function, but mpSA performs no better if $0 \in \Theta$.

We apply both the mpSA and the BASM algorithms and compare the results in Figures 4.1 and 4.2. We also discretized Θ to provide an outline of the true solution and value functions in both figures. In Figures 4.1a and 4.2a, the crosses represent x_1^* and the circles represent x_2^*; in Figures 4.1b, 4.1c, and 4.2b, the circles represent u^* or f^* respectively. For mpSA, we relax g_1 into the objective function which results in one dual solution, $u^* : \Theta \to \mathbb{R}$. We use $\lambda_k = \frac{1}{3(k+1)}$ for our stepsize and take twenty steps. Our initial $\bar{u}^k$ is the first interpolation step. Note from Figures 4.1b and 4.1c that this results in accuracy on the bounds of Θ, but, at $\theta \in (0.4, 0.8)$, $\bar{u}^k$ is not able to converge to u^* in the alloted steps. Furthermore, by this point, the stepsize is $\lambda_{20} = \frac{1}{63}$, so additional steps will be insufficient to bring the center of $\bar{u}^k(\theta)$ up by the required 3 units. That is, make it so that $\bar{u}^k(0.6) = u^*(0.6$. This behaviour is reflected in Figures 4.1a and 4.1c, which contrast with



(a) primal solutions; crosses and circles represent x_1^* and x_2^* respectively

fig:solutionSA

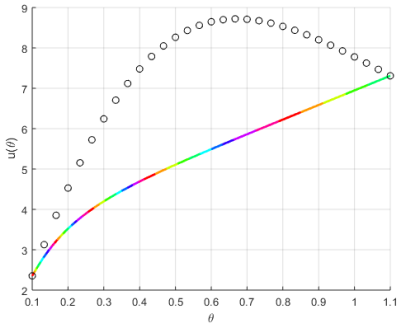


fig:dualSA

(b) dual solution; circles represent u^*

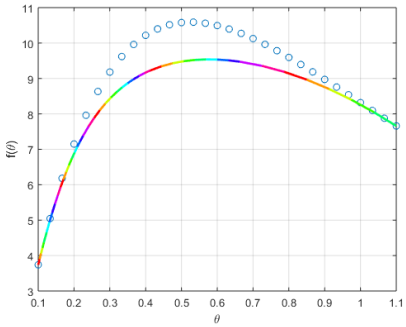


fig:valueSA

(c) value solution: circles represent f^*

fig:mpSA

Figure 4.1: The primal and dual solutions and value functions from mpSA for program (4.13).

Figure 4.2.

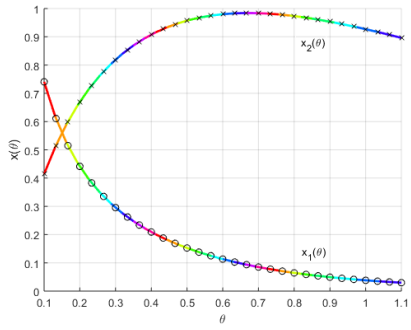
Based on this example, the limitation of mpSA to converge in a timely manner appears to be based on the curvature of $\boldsymbol{u}^*$, but this case shall be addressed in further research.

In conclusion, since the mpGQA produces unnecessarily complicated solutions, and the mpSA requires further adaptation to be properly robust, we choose the BASM as the subroutine used at each node of the BB algorithm. The BASM, being an intuitive approach, combines solid performance with the linearity of the solution functions.

4.4 Branch-and-Bound

sec:BBalgEx

Having established methods for solving continuous mpNLPs, we construct a BB algorithm with which to solve mp01NLPs and apply said algorithm to two illustrative examples, followed by consideration of randomly generated quadratically constrained instances.



(a) primal solutions; crosses and circles represent x_1^* and x_2^* respectively

fig:solutionASM

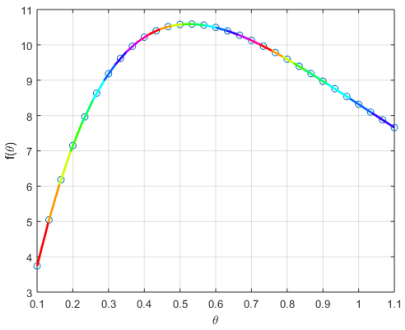


fig:valueASM

(b) value solution: : circles represent f^*

fig:BASM

Figure 4.2: The solution and value functions from the BASM for program (4.13).

4.4.1 Algorithm

Consider the solving of program (01BNLP). We revisit the relaxed problem (node) in the context of a general BB algorithm: a feasible integer solution is first found and its corresponding value function is set to be $f^{soln} : \Theta \to \mathbb{R}$, as an upper bound to which to compare the solutions found at each node. The value function at the tth node, $f^t : \Theta \to \mathbb{R}$ is compared to the current upper bound, $f^{soln} : \Theta \to \mathbb{R}$, and all simplices $\Theta_i \subset \Theta$ in which there exists $\bar{\boldsymbol{\theta}} \in \Theta_i$ such that $f^t(\bar{\boldsymbol{\theta}}) \geq f^{soln}(\bar{\boldsymbol{\theta}})$ are fathomed, that is, they are not considered when solving at subsequent children nodes. If $f^t(\bar{\boldsymbol{\theta}}) \geq f^{soln}(\bar{\boldsymbol{\theta}})$ for every $\theta \in \Theta$, then all subsequent children nodes of the tth node are removed from consideration in the algorithm. When no integer variables are relaxed to be continuous, node t is referred to as a leaf node, and $f^{soln}(\boldsymbol{\theta}) := f^t(\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in \Theta$ such that $f^{soln}(\boldsymbol{\theta}) \geq f^t(\boldsymbol{\theta})$. Different perspectives on this general algorithm as applied to parametric mixed-integer programs can be found in [85, 31, 29].

Note that there are three choices to be made arbitrarily: the subroutine with which to solve each node problem, the ordering of the nodes, and the initial node at which to solve. We choose to utilize the BASM as the subroutine, which means that $\boldsymbol{x}^{soln}$ and f^{soln} are linear interpolants of the true solutions; this has the added benefit of simplifying the fathoming step, as we need only test an extra linear constraint $f^t(\boldsymbol{\theta}) - f^{soln}(\boldsymbol{\theta}) \leq \mathbf{0}$.

At the tth node, we relax w integer-valued function variables $\mathbf{y}^t : \Theta \to [0,1]^w$, and fix the remaining $m - w$ integer-valued function variables $\mathbf{y}^\dagger : \Theta \to \{0,1\}^{m-w}$; recall from Section

4.2 that $w \in \{0, 1, \ldots, m\}$ and $\tilde{\mathbf{y}}(\boldsymbol{\theta}) = \begin{bmatrix} \mathbf{y}^{\dagger}(\boldsymbol{\theta}) \\ \mathbf{y}(\boldsymbol{\theta}) \end{bmatrix}$. Then we solve (node) with $\tilde{\mathbf{y}}(\boldsymbol{\theta}) = \begin{bmatrix} \mathbf{y}^{\dagger}(\boldsymbol{\theta}) \\ \mathbf{y}^{t}(\boldsymbol{\theta}) \end{bmatrix}$ and $\tilde{\boldsymbol{x}}(\boldsymbol{\theta}) = \begin{bmatrix} \boldsymbol{x}(\boldsymbol{\theta}) \\ \mathbf{y}^{t}(\boldsymbol{\theta}) \end{bmatrix}$. We refer to nodes at which $w = 0$ as leaf nodes; note that, for a full tree, we will have 2^m leaf nodes.

We order the nodes by counting down in ternary, with $\{relaxed, 0, 1\}$ isomorphic to $\mathbb{Z}_3 = \{0, 1, 2\}$. Note that if node t has $\mathbf{y}_1^t(\boldsymbol{\theta}) = 1$, $\mathbf{y}_2^t(\boldsymbol{\theta}), \ldots, \mathbf{y}_m^t(\boldsymbol{\theta}) \in [0, 1]^{m-1}$, then it has two children nodes, $t + 1$ with $\mathbf{y}_1^{t+1}(\boldsymbol{\theta}) = 1$, $\mathbf{y}_2^{t+1}(\boldsymbol{\theta}) = 0$, $\mathbf{y}_3^{t+1}(\boldsymbol{\theta}), \ldots, \mathbf{y}_m^{t+1}(\boldsymbol{\theta}) \in [0, 1]^{m-2}$, and $t + 2$ with $\mathbf{y}_1^{t+2}(\boldsymbol{\theta}) = 1$, $\mathbf{y}_2^{t+2}(\boldsymbol{\theta}) = 1$, $\mathbf{y}_3^{t+2}(\boldsymbol{\theta}), \ldots, \mathbf{y}_m^{t+2}(\boldsymbol{\theta}) \in [0, 1]^{m-2}$. Due to the stack we implement to traverse the tree, we choose our initial leaf node to be at $\mathbf{y}^0 = (1, 1, \ldots, 1) \in \{0, 1\}^m$.

We include the pseudocode in Algorithm 3. Begin at Step 0 by setting $w = 0$ and solving the resulting program using the BASM. This results in an initial solution $(\boldsymbol{x}^{soln}(\boldsymbol{\theta}), \mathbf{y}^{soln}(\boldsymbol{\theta}))$, with an initial value function f^{soln}, for all $\boldsymbol{\theta} \in \Theta$. At Step t, we solve the (node) corresponding to node t and compare the solution values f^t to f^{soln}. To compare the solutions, we intersect the critical regions using functions found in Multi-Parametric Toolbox [52], and compare the corresponding solution value functions. There are two different cases. If node t is a leaf, then, any time f^t is better than f^{soln}, we replace $(\boldsymbol{x}^{soln}, \mathbf{y}^{soln}, f^{soln})$ with the corresponding $(\boldsymbol{x}^t, \mathbf{y}^t, f^t)$. If node t is not a leaf, then we fathom, or remove, any critical regions where f^t is worse than f^{soln}. Then, for all children nodes and subsequent ancestors, we solve the (node) over a reduced Θ. When Θ has been reduced to $\emptyset$, we completely remove that branch. When all leaf nodes have been searched, we terminate the algorithm. A key difference between this algorithm and previous ones found in [3, 32, 75, 11] is the presence of nonlinear constraints, which result in more difficult feasibility computations. As a result we remind the reader of our assumption that each node program is feasible over all of the parameter space Θ. Additionally, due to the presence of the parameter space, and the ability of the relaxed variables to change, we do not consider integrality at branch nodes at this time.

The nonparametric BB algorithm can fathom, on average, half the nodes, resulting in a more efficient method than an exhaustive solving of all the integer combinations. However, due to the comparison of solution functions over regions as opposed to individual solution values, the parametric BB fathoms far fewer nodes. At each node, the BASM solves roughly $\kappa+1$ nonparametric nonlinear programs per simplex, where each simplex has κ vertices [77].

alg:mp-bnb

Algorithm 3: Multiparametric Branch-and-Bound

Input : Initial polyhedron, Θ over which to solve program, node subroutine

Output: Piecewise function $\boldsymbol{x}^K(\boldsymbol{\theta})$ for some $K \in \mathbb{N}$, $\mathbf{y}^K(\boldsymbol{\theta})$, and piecewise optimal function $f(\boldsymbol{x}^K, \mathbf{y}^K; \boldsymbol{\theta})$, $\boldsymbol{\theta} \in \Theta$

bnbLine0 Step 0: Obtain initial solution function $\boldsymbol{x}^{soln}(\boldsymbol{\theta})$, $\mathbf{y}^{soln}(\boldsymbol{\theta})$, $f^{soln}(\boldsymbol{\theta})$ over Θ.

Step t: At each node t, solve node problem via subroutine (BASM) to obtain linear interpolant

bnbLinet $f^t(\boldsymbol{\theta})$

Intersect critical regions from node t solution and current solution

Compare $f^t(\boldsymbol{\theta})$ and $f^{soln}(\boldsymbol{\theta})$, discarding regions where $f^{soln}(\boldsymbol{\theta})$ is better (fathoming)

if *out of nodes* **then**

| stop

else

| continue to next node, $t+1$

4.4.2 Examples

sec:BBex

We implement the aforementioned algorithm via Matlab due to the existence of the polyhedral functions found in Multiparametric Toolbox [52], a toolbox developed only in Matlab. We illustrate the algorithm with two small nonlinear, non-quadratic programs, one with one parameter, and the other with two; the two-parameter example is given in the context of an electricity demand problem for a power company. We then demonstrate the algorithm's general performance on randomly generated examples which are quadratic in $(\boldsymbol{x}, \mathbf{y})$, and affine in $\theta \in \mathbb{R}$. To keep the number of feasible regions tractable, we only let each application of the BASM split its critical regions three times.

4.4.2.1 One Parameter Illustrative NLP

Let $\Theta_1 = [0.1, 1.2]$, $\Theta_2 = [1, 4]$, $x_1, x_2, x_3 \in \mathbb{R}_{\geq}$, $y_1, y_2 \in \{0, 1\}$. Consider the following program.

$$\begin{aligned} \min_{\boldsymbol{x} \in \mathbb{R}^3_{\geqq},\, \mathbf{y} \in \{0,1\}^2} \quad & \theta {x_1}^3 + {x_2}^3 + {x_3}^3 - 2x_1y_1 - 3x_2y_2 - 4x_3y_2 \\ \text{s.t. } & \theta {x_2}^2 + y_2 \leq \theta - 1.5 \\ & x_1 + x_2 + x_3 + y_1 + y_2 \geq 3 \\ & \theta \in [1, 2]. \end{aligned} \tag{4.14}$$

3xEx

We begin the algorithm by traversing the tree in a depth-first manner, creating children

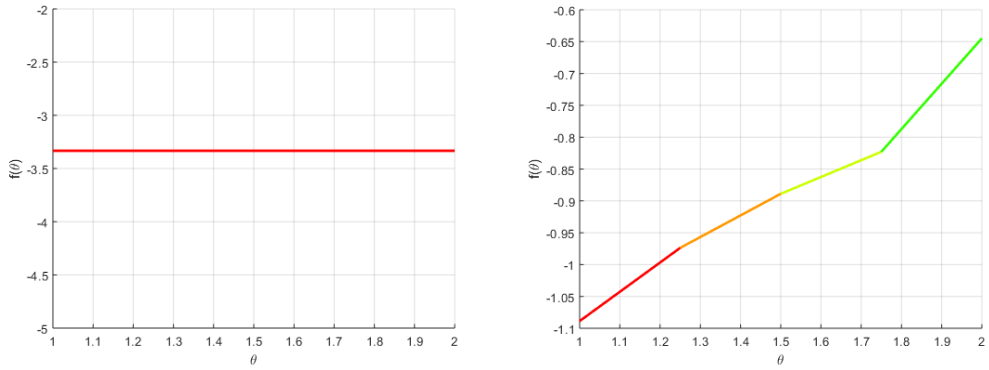


fig:3xy11

(a) Value function at $\mathbf{y} = (1, 1)$

fig:3xy10

(b) Value Function at $\mathbf{y} = (1, 0)$

Figure 4.3: The value functions at $\mathbf{y} \in \{(1, 1), (1, 0)\}$ for (4.14).

fig:3xExLeftTree

nodes as we go. As the program is feasible for all $\theta \in [1, 2]$, per our assumption at the beginning, we skip the relaxation node, and its first child, $y_1 = 1$, $y_2 \in [0, 1]$. The first leaf node we encounter is at $\mathbf{y} = (1, 1)$, whose value we denote as $f^{(1,1)}$ and display in Figure 4.3a. As we currently have no solution value function f^{soln}, we set $f^{soln} := f^{(1,1)}$, and we next check the sibling leaf node $\mathbf{y} = (1, 0)$, whose solution value function is in Figure 4.3b. Note that $f^{(1,0}(\theta) \geq f^{soln}(\theta)$ for all $\theta \in \Theta$, so we discard it. The next node is the branch node with $y_1 = 0$, $y_2 \in [0, 1]$; $f^{(0,rel)} \leq f^{soln}$ for all $\theta \geq 1.58$, but $f^{(0,rel)}(\theta) \geq f^{soln}(\theta)$, as demonstrated by comparing Figures 4.3a and 4.4a. Therefore, the next two leaf nodes are solved only over $\theta \in [1.58, 2]$. The f^{soln} gets replaced with $f^{(0,1)}$ in Figure 4.4b for $\theta \in [1.58, 2]$, but remains the same for $\theta \in [1, 1.58]$. No changes occur when comparing against $f^{(0,0)}$ in Figure 4.4c, and the final optimal value function f and final optimal solution functions $\boldsymbol{x}$ are displayed in Figure 4.5 for $\mathbf{y} = (1, 1)$ when $\theta \in [1, 1.58]$ and $\mathbf{y} = (0, 1)$ when $\theta \in [1.58, 2]$. The entire computation takes about 3 seconds.

Having considered an example in which every leaf has to be considered, we propose an example in which one of the nodes is fathomed.

4.4.2.2 Two Parameter Power Demand NLP

For the next example, consider a power company with two power generators; they need to keep their emissions below a certain amount, while meeting demand for the region, which is subject to change, as well as meeting the workers' union demands, which are static. They have the option to operate the generators in such a way as to mitigate costs (think national subsidies),

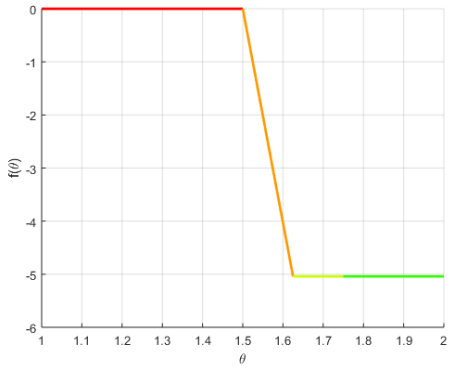


fig:3xy0inf

(a) Value Function at $y_1 = 0$, $y_2 \in [0, 1]$

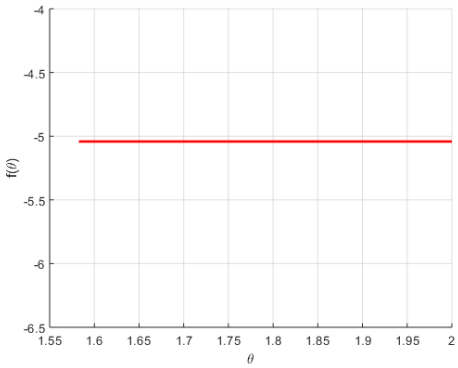


fig:3xy01

(b) Value Function at $\mathbf{y} = (0, 1)$

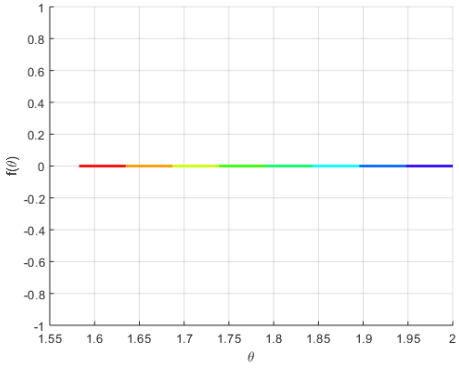


fig:3xy00

(c) Value Function at $\mathbf{y} = (0, 0)$

fig:3xExRightTree

Figure 4.4: The value functions at $y_1 = 0, y_2 \in [0, 1]$ and $\mathbf{y} \in \{(0, 1), (0, 0)\}$ for (4.14).

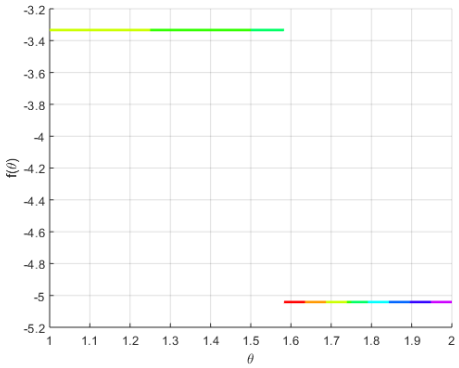


fig:3xVal

(a) Value Function for (4.14)

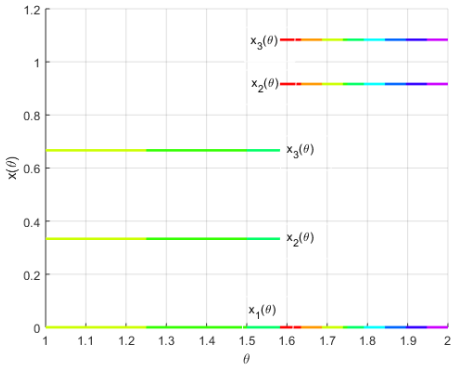


fig:3xSoln

(b) Solution Functions for (4.14)

fig:3xExFinal

Figure 4.5: The final optimal value and solution functions for (4.14) for $\mathbf{y} = (1, 1)$ when $\theta \in [1, 1.58]$ and $\mathbf{y} = (0, 1)$ when $\theta \in [1.58, 2]$.

but increase emissions and/or decrease outputs. Let $\Theta = [0.1, 1.2] \times [1, 4]$ represent the uncertainty space, $x_1, x_2 \in \mathbb{R}_{\geq}$ be the generator emissions, and $y_1, y_2 \in \{0, 1\}$ be the cost-efficient running options. Consider the following program.

$$\begin{aligned} \min_{\boldsymbol{x} \in \mathbb{R}^2_{\geqq},\, \mathbf{y} \in \{0,1\}^2} \quad & \theta_1 e^{-x_1} + 10{x_2}^2 - 3x_1 y_1 - 2(\theta_2 - 2) x_2 y_2 \\ \text{s.t. } & \theta_2 {x_1}^2 + x_1 y_2 + x_2 \leq 4 \quad \text{(emissions)} \\ & x_1 - (1 + y_1) x_2 \geq 0.5\theta_1 \quad \text{(region demand)} \\ & x_1 + x_2 \geq 1 \quad \text{(union demands)} \\ & \theta_1 \in [0.1, 1.2],\ \theta_2 \in [1, 4]. \end{aligned} \tag{4.15}$$

genEx

Note that θ_1 is an uncertainty affiliated with both the demand in the region and the cost of running generator 1, while θ_2 is an uncertainty corresponding to the environmental efficiency of generator 1 and the cost-saver mode of generator 2. Note that the union demands are requiring that the generators run for at least a certain amount so that the workers are paid.

As when solving (4.14), we begin the algorithm by traversing the tree in a depth-first manner, creating children nodes as we go. We skip the relaxation node, and its first child, $\mathbf{y}_1 = 1$, $\mathbf{y}_2 \in [0, 1]$. The first leaf node we encounter is at $\mathbf{y} = (1, 1)$, whose value we denote as $f^{(1,1)}$ and display in Figure 4.6a. As we currently have no solution value function f^{soln}, we set $f^{soln} := f^{(1,1)}$, and we next check the sibling leaf node $\mathbf{y} = (1, 0)$, which is in Figure 4.6b. Note $f^{(1,0)}(\boldsymbol{\theta}) \leq f^{soln}(\boldsymbol{\theta})$ for all $\theta_2 \geq 2$. Therefore, we replace that half of f^{soln} with the new solution.

The next node we consider is the branch node with $y_1 = 0$, $y_2 \in [0, 1]$, where $f^{(0,rel)}(\boldsymbol{\theta}) \geq f^{soln}(\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in \Theta$. As a result, the remaining children nodes, $\mathbf{y} \in \{(0, 1), (0, 0)\}$ are fathomed, and the final optimal value function f and final optimal solution functions $\boldsymbol{x}$ are displayed in Figure 4.5 for $\mathbf{y} = (1, 0)$ when $\theta_1 \in [0.1, 1.2], \theta_2 \in [1, 2]$ and $\mathbf{y} = (1, 1)$ when $\theta_1 \in [0.1, 1.2], \theta_2 \in [2, 4]$. The entire computation takes around 16 seconds, and the best option for this power company is to always have cost-saver mode for generator 1 active (that is, $y_1 = 1$), and to only keep cost-saver mode for generator 2 active when the emissions multiplier for generator 1 is $\theta_2 \geq 2$.

By computing a solution over the entire parameter space prior to the time of power generation, the company is better equipped to rapidly respond optimally to changes in demand related to weather, or unexpected events. More specifically, suppose that the demand parameter is $\theta_1 = 0.8$

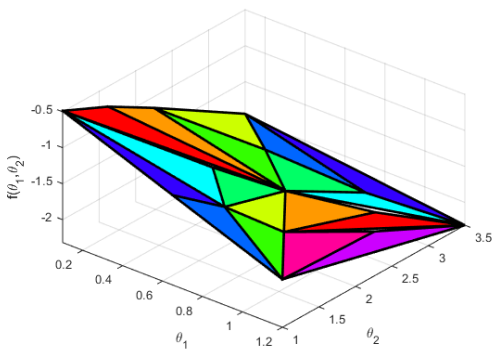


fig:geny11

(a) Value Function at $\mathbf{y} = (1,1)$

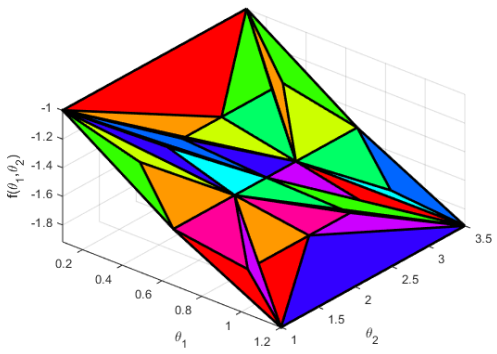


fig:geny10

(b) Value Function at $\mathbf{y} = (1,0)$

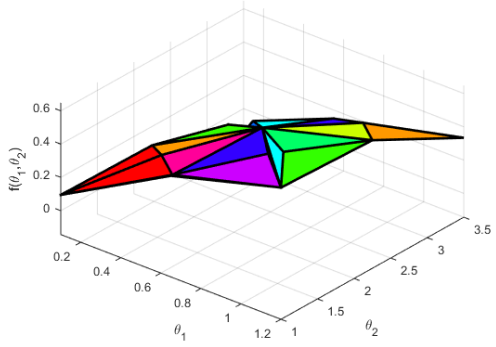


fig:geny0inf

(c) Value Function at $y_1 = 0$, $y_2 \in [0,1]$

fig:3xExNodes

Figure 4.6: The value functions at $\mathbf{y} \in \{(1,1),(1,0)\}$ and $y_1 = 0, y_2 \in [0,1]$ for (4.15).

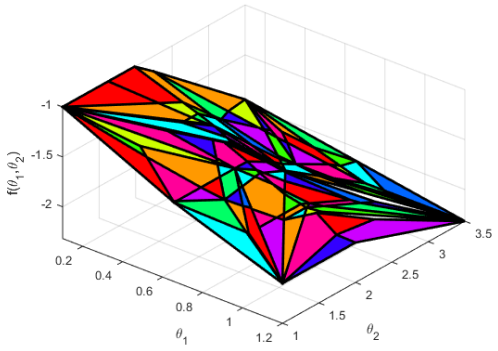


fig:genVal

(a) Value Function for (4.15)

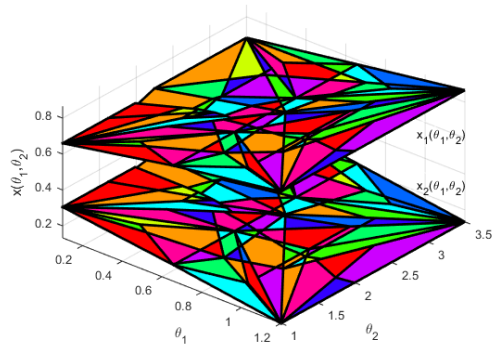


fig:genSoln

(b) Solution Functions for (4.15)

fig:genFinal

Figure 4.7: The final optimal value and solution functions for (4.15) for $\mathbf{y} = (1,0)$ when $\theta_1 \in [0.1, 1.2]$, $\theta_2 \in [1,2]$ and $\mathbf{y} = (1,1)$ when $\theta_1 \in [0.1, 1.2]$, $\theta_2 \in [2,4]$.

and the efficiency parameter for generator x_1 is $\theta_2 = 2.7$. Then the critical region corresponding to this value is

$$CR = \left\{ (\theta_1, \theta_2) : \begin{bmatrix} 0.8740 & -0.3845 \\ -0.3229 & 0.4011 \\ -0.9102 & 0.1024 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \leq \begin{bmatrix} -0.2971 \\ 0.8573 \\ -0.4013 \end{bmatrix} \right\}.$$

Note that, by the previous analysis, the integer decision variables are $\mathbf{y} = (1, 1)$. The optimal continuous solution function on this region, for the aforementioned integer variables,

$$\bar{\boldsymbol{x}}(\theta_1, \theta_2) = \begin{bmatrix} 0.1667\theta_1 - 0\theta_2 + 0.6667 \\ -0.1667\theta_1 - 0\theta_2 + 0.3333 \end{bmatrix},$$

has approximate value function $\bar{f}(\theta_1, \theta_2) = -0.1518\theta_1 - 0.4569\theta_2 - 0.5366$. Thus, at uncertainty $(0.8, 2.7)$, both cost-saver modes are active, and the runtime for the generators is $\bar{\boldsymbol{x}}(0.8, 2.7) = (0.8, 0.2)$ units with an approximate cost (benefit with the negative sign) of $\bar{f}(0.8, 2.7) = -1.8916$ units.

As another example, consider the uncertainty parameters $\theta = (0.4, 1.8)$. Then the containing critical region is

$$CR = \left\{ (\theta_1, \theta_2) : \begin{bmatrix} 0.8740 & -0.3845 \\ 0.1820 & 0.3893 \\ -0.6846 & 0.4511 \\ -0.9257 & 0.3097 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \leq \begin{bmatrix} -0.2972 \\ 0.9030 \\ 0.5725 \\ 0.2171 \end{bmatrix} \right\}.$$

Since $\theta_2 < 2$, the integer decision variables are $\mathbf{y} = (1, 0)$. The optimal continuous solution function on this region, for the aforementioned integer variables,

$$\bar{\boldsymbol{x}}(\theta_1, \theta_2) = \begin{bmatrix} 0.0838\theta_1 + 0.0546\theta_2 + 0.5973 \\ -0.2081\theta_1 + 0.2732\theta_2 + 0.2987 \end{bmatrix},$$

has approximate value function $\bar{f}(\theta_1, \theta_2) = -0.7812\theta_1 - 0.0726\theta_2 - 0.8474$. Thus, at uncertainty $(0.4, 1.8)$, only the first cost-saver mode is active, and the runtime for the generators is $\bar{\boldsymbol{x}}(0.4, 1.8) = (0.7292, 0.2646)$ units with an approximate cost (benefit with the negative sign) of $\bar{f}(0.4, 1.8) = -1.2905$ units.

Note that, due to unknown interactions such as subsidies etc., a negative objective value function can make sense, even in this application.

4.4.2.3 Random QCQPs

For the random instances, we consider $\boldsymbol{x} \in \mathbb{R}^3$, $\theta \in [0.1, 1.1]$, and $y \in \{0,1\}^m$ with $m \in \{2,3,4\}$, with one quadratic constraint and two linear constraints; the format of the program is as follows.

$$\begin{aligned}
\min_{x\in\mathbb{R}^3,\, \mathbf{y}\in\{0,1\}^m} \quad & \begin{bmatrix} \boldsymbol{x}^T & \boldsymbol{x}^T & \mathbf{y}^T & \mathbf{y}^T \end{bmatrix} Q(\theta) \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \\ \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} + \boldsymbol{q}(\theta)^T \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} \\
\text{s.t.} \quad & \begin{bmatrix} \boldsymbol{x}^T & \boldsymbol{x}^T & \mathbf{y}^T & \mathbf{y}^T \end{bmatrix} Q_{con}(\theta) \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \\ \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} + \boldsymbol{q_{con}}(\theta)^T \begin{bmatrix} \boldsymbol{x} \\ \mathbf{y} \end{bmatrix} + \boldsymbol{c}(\theta) \leq 0, \qquad (4.16) \quad \texttt{randEx} \\
& A_x \boldsymbol{x} + A_y \mathbf{y} \leq \boldsymbol{b} + E\theta, \\
& \theta \in [0.1, 1.1].
\end{aligned}$$

Note that $Q, Q_{con} : [0.1, 1.1] \to \mathbb{R}^{(3+m)\times(3+m)}$ are constructed to be positive semidefinite as follows: $Q(\theta) = P(\theta)^T P(\theta)$ and $Q_{con}(\theta) = P_{con}(\theta)^T P_{con}(\theta)$, where $P, P_{con} \in \mathbb{R}^{(3+m)\times(3+m)}$ are generated from a standard normal distribution with each element having a fifty percent chance to have θ. The vectors $\boldsymbol{q}, \boldsymbol{q}_{con} : [0.1, 1.1] \to \mathbb{R}^{3+m}$ are defined similarly to P and P_{con}. The other matrices $A_x \in \mathbb{R}^{2\times 3}$, $A_y \in \mathbb{R}^{2\times m}$ and E are generated from a standard normal distribution as well, while $\boldsymbol{b}$ and $\boldsymbol{c}(\theta)$ are generated in such a way that there will be a feasible point for every $\theta \in [0.1, 1.1]$. Note that this method is based on that found in [26].

For each t, we solve ten randomly generated instances; in Table 4.1, we record the mean, median, and standard deviations of time (in seconds), nodes solved, number of simplices stored, and average error per simplex. The computation of this error is from the BASM (see Section 4.3.2 and [77]).

integers	statistics	time (sec)	nodes	polytopes	error
	mean	4.0373	4.6	23.1	0.0223
2	median	4.4268	5	26	0.0146
	std. dev.	1.3832	0.8433	7.1562	0.0209
	mean	20.4567	11	63.8	0.0383
3	median	20.0576	12	56.5	0.0202
	std. dev.	9.4150	1.9437	27.0259	0.0503
	mean	201.0893	21.2	866	0.1024
4	median	57.0664	19	206.5	0.0618
	std. dev.	310.6684	3.5528	146.9999	0.1319

Table 4.1: Comparison of average errors, CPU times, and number of simplices with three continuous variables, one quadratic constraint, two linear constraints, one parameter, at ten instances each.

tab:exErrors

As seen in Table 4.1, three-quarters of the nodes, branch and leaf, have to be evaluated. This differs from the nonparametric versions due to the fact that there is now a region to be compared, rather than a single point. Therefore it is much more likely that there will be a location at which the current node is better. Keep in mind that m branch nodes leading to the initial feasible solution are always omitted due to our implementation. Additionally, the number of critical regions grows swiftly even for only one parameter. We also note that, based on the mean, median, and standard deviation for $m = 4$, there are one or two instances which are much larger in scope than the others; this skews the error upwards. However, looking only at the median, we see that the error seems to scale with m slower than the other data. As intersections of the critical regions currently increase quadratically with this number, we opt not to test instances for parameter space $\Theta \subseteq \mathbb{R}^2$.

This is the first algorithm to solve nonlinear programs with parameters in general locations, so we do not have other methods with which to compare. Additionally, there is very little random benchmarking for parametric programs in the literature, which, again, limits our ability to compare. Therefore, we refrain from drawing any conclusions as to the efficacy of the algorithm.

4.5 Conclusion

sec:con

Given the current state-of-the-art in parametric programming, we constructed a branch-and-bound algorithm for mixed-binary nonlinear programs, a first of its kind for its generality. We introduced the topic based on its potential applications in both multiobjective programming and more applied settings that require a finer resolution than the worst-case response. To further cement this, we demonstrated its potential application in a simple but non-polynomial power-generation problem. We then applied the algorithm to a set of randomly generated instances to provide foundations for benchmarking of future algorithms. We notice that the number of polytopes stored in our algorithm is constrainingly large, so future work would include getting around that.

There are several options open to us at present: the closest related would be to implement a merge routine in which we collate the critical regions based on their variable coefficients, and then merge said critical regions; next would be to investigate the implementation of a feasibility computation along the lines of that found for linear constraint methods in order to increase fathoming; to avoid BB via cutting algorithms, so as to eliminate the need to compare at each node. Additionally, our benchmarking randomly placed the parameter with no limit to how many places it could be; breaking down the analysis by parameter would be an additional goal.

Chapter 5

Conclusions and Discussion

5.1 Contributions

We expanded parametric programming in a number of different ways. We considered the application of approximate parametric programming algorithms to multiobjective programming, and with that notion in hand, we introduced additional scalarization methods. Building on those methods, we then addressed optimization models with nonlinear functions for both continuous and mixed-integer variables. Throughout this work, we illustrated the algorithms and their performances via numerical experiments previously unknown in parametric programming literature.

Though it does not appear until Chapter 4, we actually began our work by exploring the construction of a branch-and-bound algorithm utilizing a subgradient ascent algorithm as the continuous subroutine and discovering the impracticability thereof, shown in Section 4.3.3. As a result, we turned our attention instead to the implications that parametric programming, specifically a method involving linear interpolation over the parameter space, has for multiobjective programming. We began by expanding the list of possible scalarization methods, and followed up by comparing an analytic linear complementarity method constructed by our predecessors [5] against an approximation method [14] on a variety of both constructed and randomly generated examples. Through this exploration of MOPs, we discovered that the literature for parametric programming largely omits the differentiation between joint and bi- convexity (between the variables and parameters) and contents itself with the term convexity, meaning the more stringent joint convexity.

Due to the frequent appearance of bilinear terms in applications such as pooling problems,

we investigated the difference between joint and bi- convexity (still between the variables and parameters) as it applies to linear interpolation. We discovered that care had to be taken when constructing error bounds, since the proof of convergence for jointly convex objectives and constraint functions no longer work when extended to biconvex functions. We combed both the parametric optimization and interpolation literature banks for an adaptive bound for linear interpolation, but were unable to find one. As such, we constructed an upper bound ourselves based on the biconvex analog to the convex overestimator. To confirm that we were not retreading old ground, we checked the literature for computational analysis and the methodology of picking new interpolation points for adaptive meshes. In so doing, we discovered the link between our optimization problem and problems of variation: the literature [21] indicates that, for problems of variation, there are no upper bounds on the error; instead, adaptive meshes work to reduce the lower bound on error terms. Since we have an upper bound that we can reduce, we are able to guarantee convergence, while the standard lower bound reduction can only hope for it; this new guarantee is naturally a good thing. We compared the runtimes of this adaptive mesh with a simpler, more static method of choosing interpolation points as a demonstration of our error bounds usefulness.

Having constructed this biconvex approximation, we then returned to our original plan of a branch-and-bound algorithm, this time with our new linear interpolation method as the subroutine. We briefly exposit the possibility of using a quadratic approximation (also extended to consider functions biconvex in the variables and parameters), as well as demonstrated the difficulties entwined in using the formerly considered subgradient ascent algorithm. We concluded by constructing the branch-and-bound algorithm and demonstrating its possible use in applications such as power generation. As with our other work, we also included a set of benchmarks for comparison with later algorithms. As mentioned at the beginning of this section, benchmarking for randomized instances of similar type is absent in parametric programming literature; as such, we expanded on work found in [26] and provide it in the Appendix for comparison by future researchers.

5.2 Further Research

A lot of further research entails a strengthening of the benchmarks which we found to be absent from the literature: increasing the number of variables, paying closer attention to the locations of parameters in programs, comparing different optimization functions as subroutines, and the like.

Checking the threshold at which parametric programming in its initial computational complexity becomes less resource-intensive than individual optimizing or other forms of optimization under uncertainty would also be of interest.

On the topic of mixed integer programming, the branch-and-bound algorithm we constructed would benefit from certain modifications and adjustments: introducing a method to trim infeasible regions from the parameter space would broaden the number of applicable situations; another beneficial adjustment would be the introduction of a merge function which combines invariancy regions which are adjacent and share the same interpolations, and therefore can be combined. This reduction in the number of invariancy regions would decrease the amount of time spent comparing intersections. Alternatively, inducing additional constraints to cut out previously obtained integer solutions or optimal fractional solutions, i.e. applying a cutting algorithm, would also be worth considering.

A broader scale considers the hybridization of parametric programming with other subfields of optimization, similar to that done in multiobjective programming. For instance, it may be worth considering stochastic optimization by interchanging a random variable or two from the program with parameters and considering the results. Another option would be to look at incorporating a parametric program within a machine learning model so that part of its optimization is checking a table for a function to evaluate, rather than an entire optimization.

Appendices

Before or while trying to figure out any of the code below, it is recommended that the interested user check the MATLAB documentation on the *fmincon* function, as well as referring to [52] for any of the polyhedron or set operation functions. It may also be helpful to see the documentation on the use of *varargin*. More generally, the strange words in the code variables, such as 'wah,' 'potate,' or, in one instance, 'breadFox,' are references to the author's parents' cats; we needed names to differentiate, and those were the first to come to mind.

Appendix A Random Functions

sec:randFun

The following code is the set of random functions used in constructing the benchmark numbers; *makeConstraints.m*, and its updated successor *makeConstraints3params.m* in C are to construct the nonlinear constraints file to be used anytime the black-box function *fmincon* is called; use of the one versus the other really only depends on when each was written.

```
function [] = makeConstraints(constraints,twoParams, numVars, nameIndex)
%create the constraints file that can then be read by fmincon
%-------------------------------------------------------------------------
%INPUT          TYPE                    DESCRIPTION
%constraints:   cell                    quadratic constraints
%twoParams:     boolean                 are there two parameters
%numVars:       double                  number of variables
%nameIndex:     double                  which example this is
%-------------------------------------------------------------------------
numCons=length(constraints);
charIdx=num2str(nameIndex);
fName=['nonlcon' num2str(numVars) 'VarsEx' charIdx '.m'];
fileID=fopen(fName,'w');

%note that matlabFunction calls a symbolic vector 'in#' where #represents
%the index at which it is located as soon as the symbolic vector is larger
%than 1x1. For instance, if theta1,theta2\in R and x\in R^2, then
%{theta1,theta2,x} becomes {theta1,theta2,in3} while {x,theta1,theta2}
%becomes {in1,theta1,theta2}
```

```
%technically, we should obtain where the vectors are located in the
%constraints, but I'm lazy: the code is forcing it to be 'in3' so that's
%what we'll make it
temp=['function [g,h] = nonlcon' charIdx];
if twoParams
    firstLine=[temp '(theta1,theta2,in3)\n'];
else
    firstLine=[temp '(theta1,in2)\n'];
end

fprintf(fileID, firstLine);
for i=1:numCons
    %find the first instance of ')' and then delete all parts of the array
    %prior to it; basically the '@(<argin>)'
    temp=char(constraints{i});
    closeParenIdx=strfind(temp,')');
    temp=temp(closeParenIdx(1)+1:end);
    constLine=['g(' num2str(i) ')=' temp ';\n'];
    fprintf(fileID, constLine);
end %for

%modify to accept equality constraints at some point
fprintf(fileID, 'h=[];\n');
fprintf(fileID, 'end');
fclose(fileID);

end %function
```

```
function binMat = rand01(size,percentOnes)
binMat=rand(size);
temp=1-percentOnes;
binMat=binMat>temp;
end %function
```

```
function outMat = randMat(size, varargin)
if isempty(varargin)
    mean=0;
    stdDev=1;
else
    mean=varargin{1};
    stdDev=varargin{2};
end %if
outMat=mean*ones(size)+randn(size)*stdDev;
end %function
```

Appendix B Chapter 2 Code

sec:MOPcode

B.1 Approximate Simplex Method [14]

This version of the Approximate Simplex Method is largely the same as that found in [60]; it was created, as the name suggests, prior to our consideration of the difference between joint convexity and biconvexity. Note that these two functions need to be stored in the same folder as *mpASM_preLEM* calls *mpSimplexApproximation_preLEM*.

```
function [xstar,fstar,CR,solnInfo] = mpASM_preLEM(tol,Omega,problem,varargin)
%This function solves a general mp-NLP using the simplex approximation
%algorithm of Bemporad and Filippi (2006)
%-------------------------------------------------------------------------
%INPUT          TYPE            DESCRIPTION
%tol:           scalar          error tolerance
%Omega:         polyhedron      parameter space
%problem:       struct          mp-NLP information
%varargin:      cell: scalar    optional cap for CR splits
%               cell: chars     optional error type: 'BOM'
%                                                    'LEM'
%-------------------------------------------------------------------------
%OUTPUT
```

```
%xstar:                 cell                optimal decision function
%fstar:                 cell                optimal value function
%CR:              polyhedron vector         partition of parameter space
%solnInfo:              struct              solution information
%-------------------------------------------------------------------------
%solnInfo Fields:
%error:      sum of largest single point errors for each simplex
%intError:   approximation of integrated error over parameter space
%nlps:       total number of NLP problems solved
%tol:        maximum error tolerance used
%numTooSmallPolys:      total number of Polyhedra that were not splittable
%time:       number of milliseconds to run
%-------------------------------------------------------------------------
tic
if isempty(varargin)            % Pangia 6-2-2020
    cap=5;
else
    cap=varargin{1};
end %if
%Divide parameter space into simplices
S = Omega.triangulate;
%=========================================================================
%Construct a linear approximate of the solution for each simplex
RemainVolume = Omega.volume;

[xstar,fstar,CR,solnInfo,~] = mpSimplexApproximation_preLEM(tol,S,problem,...
                                RemainVolume,cap);
solnInfo.tol = tol;
solnInfo.time = toc;


%=========================================================================
end
```

```
1  function [xOpt,fOpt,CRopt,metric,RemainVol] = ...
2                      mpSimplexApproximation_preLEM(tol,S,problem,RemainVol,cap)
3  %Constructs a linear approximation of the optimal decision and value
4  %functions for each simplex in S. Calculates the maximum error
5  %over each simplex and partitions it further if error is too large.
6  %-------------------------------------------------------------------------
7  %INPUT          TYPE                    DESCRIPTION
8  %tol:           scalar                  error tolerance
9  %S:             polyhedron array        set of simplexes
10 %problem:       struct                  mp-NLP information
11 %RemainVol:     scalar                  remaining parameter space to check
12 %cap:           scalar                  max number of CR splits
13 %-------------------------------------------------------------------------
14 %OUTPUT
15 %xOpt:          cell                    optimal decision function
16 %fOpt:          cell                    optimal value function
17 %CRopt:         polyhedron vector       partition of parameter space
18 %metric:        struct                  solution information
19 %RemainVol:     scalar                  remaining parameter space to check
20 %-------------------------------------------------------------------------
21 %get number of parameters and variables
22 %=========================================================================
23 pn = S(1).Dim;
24
25 xn = length(problem.lb);
26 %=========================================================================
27 %initialize struct fields, cells, and arrays
28 %=========================================================================
29 metric.error = 0;
30 metric.intError = 0;
31 metric.nlps = length(S)*(pn+2);
32 metric.xError = 0;    %Pangia 3-4-2020
33 metric.numTooSmallPolys = 0;    %Pangia 5-28-2020
```

```
34 xOpt = cell(0);
35 fOpt = cell(0);
36 CRopt = [];
37 %==========================================================
38
39 for i = 1:length(S)
40     vertices = S(i).V;
41     CRvol = S(i).volume;
42     M = [vertices, ones(pn+1,1)];
43     X = zeros(xn,pn+1);
44     fval = zeros(1,pn+1);
45
46     %determine optimal solution at each vertex
47     int0 = problem.lb;
48     for j = 1:pn+1
49         [X(:,j),fval(j)] = optimizationProblem(int0,vertices(j,:)',problem);
50     end
51
52     %linear approximations of optimal value function and decision functions
53     %Value Function:          zApprox = fval*(M^-1)[1;t]
54     %Decision Function:       xApprox = X(M^-1)[1;t]
55 %      Minv = M^-1;
56 %      xApprox = (Minv*X')';
57 %      zApprox = (Minv*fval')';
58
59     %1-7-21: note that xApprox=X*M^-1 is the same as solving xApprox*M=X
60     %equivalently, xApprox'=Minv*X' is the same as solving M*xApprox'=X'
61     % which is done by computing xApprox=(M\(X'))'
62     %similarly, compute zApprox=(M\(fval'))'
63     xApprox=(M\(X'))';
64     zApprox=(M\(fval'))';
65
66     %==========================================================
```

```
67      %determine maximum error
68      %=========================================================
69      %computes maximum error for zApprox as well as
70      %evaluating f at the linearly interpolated optimizers based on xApprox
71      %see Bemporad 2006 paper for more details
72  %%-----------------------------------------------------------
73      %Error Type 1: max |zApprox(t) - f*(t)|
74      tCenter = mean(vertices)';
75
76      x0 = [xApprox*[tCenter;1];tCenter];
77      %note that tHat is being computed to not necessarily be in the simplex
78      [xHat,tHat,error1] = computeError(problem,S(i),x0,zApprox);
79
80
81
82      %For nonconvex cases: Check error at midpoint if tHat is a vertex:
83      %1-14-21: consider if ~S(i).contains(tHat) then
84      %tHat=project(S(i),tHat)
85      tM = repmat(tHat',pn+1,1);
86      vertDistance = min(sum(abs(tM-vertices),2));
87
88      if vertDistance <= pn*(1e-5)
89          %6-25-2021: make the split point be the center of the longest edge to
90          %ensure no squat triangles (due to error bounds for interpolation)
91          maxLength=Inf;
92          leftVertInd=0;
93          rightVertInd=0;
94          for moo=1:length(vertices)
95              for wah=moo+1:length(vertices)
96                  edgeLength=norm(vertices(moo,:)-vertices(wah,:));
97                  if edgeLength<maxLength
98                      maxLength=edgeLength;
99                      leftVertInd=moo;
```

```
100                         rightVertInd=wah;
101                     end
102                 end
103             end
104             %center of longest edge
105             tCenter=((vertices(leftVertInd,:)+vertices(rightVertInd,:))/2)';
106
107             tHat=tCenter;
108             x0 = xApprox*[tHat;1];
109             [xHat,fHatb] = optimizationProblem(x0,tCenter,problem);
110             error1 = abs(fHatb-zApprox*[tHat;1]);
111         end
112 %%-------------------------------------------------------------------------
113         %Error Type 2: max |f(xApprox(t)) - f*(t)|
114         xBar = xApprox*[tHat;1];
115         error2 = abs(problem.objective(xHat,tHat) - ...
116                     problem.objective(xBar,tHat));
117 %%-------------------------------------------------------------------------
118         %=========================================================
119         %Choose which type of error should be used for determining the quality
120         %of the solution:
121         %errorType = error2;
122
123         errorType = error1;
124
125 %Pangia 4-25-2020 commented these out
126 %       disp('**************************************************')
127 %       disp(['Largest error at t = ' num2str(tHat')])
128 %       disp(['zApprox error is ' num2str(error1)])
129 %       disp(['f(xApprox) error is ' num2str(error2)])
130 %       disp(['Region Volume is ' num2str(CRvol)])
131 %       disp(['Total Volume Remaining is ' num2str(RemainVol)])
132 %       disp('**************************************************')
```

```
133 %     disp('  ')
134
135     S2 = splitSimplex(S(i),tHat);
136
137     %if the polyhedron S(i) is too small to split (assume it is not)
138     canSplit=~isempty(S2);     %Pangia 5-12-2020
139
140     if (errorType < tol || ~canSplit || cap <= 0)
141         %Solution is acceptable, save and move to next simplex;
142         %alternatively, solution is unacceptable, but we can't fine-tune it
143         %anymore, or the number of splits has been exceeded
144         xOpt = [xOpt;xApprox];
145         fOpt = [fOpt;zApprox];
146         CRopt = [CRopt;S(i)];
147         metric.error = metric.error + errorType;
148         metric.intError = metric.intError + errorType*CRvol;
149         metric.xError = metric.xError + error2;     %Pangia 3-4-2020
150         metric.numTooSmallPolys=metric.numTooSmallPolys+~canSplit; %Pangia
                5-29-2020
151         RemainVol = RemainVol - CRvol;
152     else
153         %Error is too large, split simplex and make new solution
154
155         [x2,f2,CR2,metric2,RemainVol] = mpSimplexApproximation_preLEM(...
156                                             tol,S2,problem,RemainVol,cap-1);
157         metric.error = metric.error + metric2.error;
158         metric.xError= metric.xError + metric2.xError;    %Pangia 3-5-2020
159         metric.intError = metric.intError + metric2.intError;
160         metric.nlps = metric.nlps + metric2.nlps;
161         metric.numTooSmallPolys=metric.numTooSmallPolys+...
162             metric2.numTooSmallPolys;
163         xOpt = [xOpt;x2];
164         fOpt = [fOpt;f2];
```

```
165             CRopt = [CRopt;CR2];
166         end %if
167 end %for
168 end
169
170
171 function [xHat,tHat,error] = computeError(problem,S,x0,zHat)
172 SA = S.A;
173 Sb = S.b;
174 m = size(SA,1);
175 n = length(problem.lb);
176
177 A = [problem.A, -problem.E;
178       zeros(m,n),SA];
179 b = [problem.b;Sb];
180
181 Aeq=[problem.Aeq, -problem.Eeq];
182 beq=problem.beq;
183
184 lb = [problem.lb;min(S.V)'];
185 ub = [problem.ub;max(S.V)'];
186
187 obj = @(x)nlpObjective(x,n,zHat,problem.objective);
188 if isempty(problem.constraints)
189     const = [];
190 else
191     const = @(x)nlpConstraints(x,n,problem.constraints);
192 end
193
194 % options = optimoptions('fmincon','Algorithm','interior-point','Display','Off
        ');
195 options = optimoptions('fmincon','Algorithm','sqp','Display','off');
196 [x,error] = fmincon(obj,x0,A,b,Aeq,beq,lb,ub,const,options);
```

```
197 error = abs(error);
198 tHat = x(n+1:end);
199 xHat = x(1:n);
200 end
201
202 function f = nlpObjective(u,n,z,func)
203 %objective function for mp-nlp with parameters treated as variables
204 x = u(1:n);
205 p = u(n+1:end);
206 %note that this works only for jointly convex functions
207 f = func(x,p)-z*[p;1];
208
209 end
210
211 function [c,ceq] = nlpConstraints(u,n,func)
212 %Nonlinear constraints for mp-nlp with parameters treated as variables
213 x = u(1:n);
214 p = u(n+1:end);
215 [c,ceq] = func(x,p);
216 end
```

sec:MOPbench

B.2 Benchmark Functions

These benchmark functions for 2.4.2 need to point to where the random functions in Section A are stored; this explains the *addpath* function calls. If this benchmarking is going to be mimicked, the benchmarking files in Section C.2 are the most well-done.

```
1 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      Matlab codes\mp ASM algorithm\mpASM')
2 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      MPT Practice\mpSAPractice')
3
4 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      MPT Practice\mpSAPractice\randFuncs')
```

```
5
6  %Random Example (plots)
7  %three variables
8  % numX=2;
9  % numX=3;
10 % numX=4;
11 numX=5;
12 % xSize=[1,numX];      %row form;
13
14 %MUST BE COLUMN FORM RIGHT NOW
15 xSize=[numX,1];       %col form;
16
17 x=sym('x',xSize);
18 assume(x, 'real');
19
20 %one parameter
21 syms theta1;
22 assume(theta1>=0 & theta1<=1);
23 % assume(theta2>=0 & theta2<=1);
24
25 %ten examples, with 1 nonlinear constraint each
26 numExamples=10;
27 numNonlinConstraints=1;
28
29 %the parameter space
30 Theta = Polyhedron('lb',[0],'ub',[1]);
31
32 %depth for mp-ASM
33 cap=5;
34
35 %number of dual updates for mp-SA
36 maxIter=10;
37
```

```
38 %initial guess for dual variables: v does not exist
39 %Note u0={zeros(1,(1+numParams))};
40 u0={zeros(1,2)};
41 v0={};
42
43 %set the precision of the symbols
44 digits(5);
45
46 %set the seed so that there is some semblance of replicability
47 rng default;
48 seedNum=8+numX;
49 rng(seedNum);
50
51 %set the guaranteed feasible point
52 % bSelect=ones(xSize);       %if x is a row
53 bSelect=ones(fliplr(xSize));         %if x is a col
54
55 %Set the linear constraint Ax + E*theta<=b
56 numLinCon=1;
57 Asize=[numLinCon,numX];
58 Esize=[numLinCon,1];    %1 being for theta1
59 cSize=[1,numLinCon];
60
61 %the theta-picking matrix for the linear matrix
62 RaSize=[numLinCon,numLinCon];
63 RcSize=[numLinCon,numLinCon];
64
65 %the matrices for generating the objective function and the quadratic
66 %constraints
67 %Note Q=P'P
68 Psize=[numX,numX];
69 RpSize=Psize;
70 qSize=[1,numX];
```

```
71
72  cQuadSize=[1,1];
73  RcQuadSize=[1,1];
74
75  mpsaEx=cell(1,numExamples);
76  mpsaExTemp=struct;
77
78  %solution fields
79  xSA=cell(1,numExamples);
80  uSA=cell(1,numExamples);
81  vSA=cell(1,numExamples);
82  fSA=cell(1,numExamples);
83  CRSA=cell(1,numExamples);
84  solnInfoSA=cell(1,numExamples);
85
86  xASM=cell(1,numExamples);
87  fASM=cell(1,numExamples);
88  CRASM=cell(1,numExamples);
89  solnInfoASM=cell(1,numExamples);
90
91  %the constraints
92  gurke=cell(numExamples,numNonlinConstraints);
93  constraintsList=cell(numExamples,numNonlinConstraints);
94
95  %the objectives
96  objList=cell(numExamples,1);
97
98  for i=1:numExamples
99      disp(['Creating Example ' num2str(i) ' of ' num2str(numExamples)]);
100     mpsaExTemp.A=floor(randMat(Asize,0,1)*1000)/1000;
101     mpsaExTemp.E=floor(randMat(Esize)*rand01([1,1],0.5)*1000)/1000;
102     %[1,1] being for theta1
103     mpsaExTemp.b=mpsaExTemp.A*bSelect';
```

```
104     mpsaExTemp.Aeq = [];
105     mpsaExTemp.Eeq = [];
106     mpsaExTemp.beq = [];
107
108     %unrelaxed nonlinear constraints
109     mpsaExTemp.baseConstraints=[];
110
111     %make the objective function
112     %Q=P'(theta)*P(theta)
113     %P(theta)=P1+P2*Rp*theta
114     P1=floor(randMat(Psize,0,1)*1000)/1000;
115     P2=floor(randMat(Psize,0,1)*1000)/1000;
116
117     Rp=floor(rand01(RpSize,0.5)*1000)/1000;
118
119     %q'(theta)*x
120     q1=floor(randMat(qSize,0,1)*1000)/1000;
121     q2=floor(randMat(qSize,0,1)*1000)/1000;
122     Rq=floor(rand01(RpSize,0.5)*1000)/1000;
123
124     P=vpa(P1+P2*Rp*theta1);
125     Q=expand(P'*P);
126     q=vpa(q1+q2*Rq*theta1);
127
128 %       obj=x*Q*x'+q*x';      %if x is a row
129       obj=vpa(x'*Q*x+q*x);      %if x is a col
130       objList{i}=vpa(x'*Q*x+q*x);
131       obj=matlabFunction(obj,'Vars',{theta1,x});
132
133       %objective function; inner variables must be different than syms
134       mpsaExTemp.objective=@(varX,varTheta) ...
135       obj(varTheta(1),varX);
136
```

```
137     for j=1:numNonlinConstraints
138         %Q=P'(theta)*P(theta)
139         %P(theta)=P1+P2*Rp*theta
140         P1=floor(randMat(Psize,0,1)*1000)/1000;
141         P2=floor(randMat(Psize,0,1)*1000)/1000;
142         Rp=floor(rand01(RpSize,0.5)*1000)/1000;
143
144         %q'(theta)*x
145         q1=floor(randMat(qSize,0,1)*1000)/1000;
146         q2=floor(randMat(qSize,0,1)*1000)/1000;
147         Rq=floor(rand01(RpSize,0.5)*1000)/1000;
148
149         %c(theta)
150         cQuad1=floor(randMat(cQuadSize,0,1)*1000)/1000;
151         cQuad2=floor(randMat(cQuadSize,0,1)*1000)/1000;
152         RcQuad=floor(rand01(cQuadSize,0.5)*1000)/1000;
153
154         P=vpa(P1+P2*Rp*theta1);
155         Q=expand(P'*P);
156         q=vpa(q1+q2*Rq*theta1);
157         cQuad=vpa(cQuad1+cQuad2*RcQuad*theta1);
158
159
160         %b=bSelect'*Q(theta)*bSelect+q'(theta)*bSelect+cQuad(theta)
161         b=bSelect*Q*bSelect'+q*bSelect'+cQuad;
162
163         %generate the nonlinear constraints for mp-SA
164 %           gurke{j}=x*Q*x'+q*x'+cQuad-b;          %if x is a row
165         gurke{i,j}=vpa(x'*Q*x+q*x+cQuad-b);     %if x is a col
166         constraintsList{i,j}=gurke{i,j};
167         gurke{i,j}=matlabFunction(gurke{i,j},'Vars',{theta1,x});
168         %constraints to be relaxed; inner variables must be different than
169         %syms
```

```
170             mpsaExTemp.relConIneq{j}=@(varX,varTheta) ...
171                 gurke{i,j}(varTheta(1),varX);
172         end
173         %create the nonlcon constraint for mp-ASM
174         makeConstraints(gurke(i),0,numX,i);
175
176         mpsaExTemp.relConEq=cell(0);
177
178         %lower and upper bounds for x
179         %WARNING: need to be columns for mpASM
180         boundDim=xSize;
181         mpsaExTemp.lb = zeros(boundDim);
182         mpsaExTemp.ub = inf*ones(boundDim);
183
184         mpsaEx{i}=mpsaExTemp;
185 end
186
187 %compute mp-ASM
188 for i=1:numExamples
189         %compute the example with the proper constraints
190         filename=['nonlcon' num2str(numX) 'VarsEx' num2str(i) '.m'];
191         copyfile(filename, 'nonlcon.m');
192         mpsaEx{i}.constraints=@(x,theta) nonlcon(theta(1),x);
193
194         [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
195             mpASM(0.01,Theta,mpsaEx{i},cap);
196
197         %after computing, draw the graphs of the functions that have poor error
198         %values
199         if(1)
200             %make a grid pattern of the actual solution function
201             numPoints=21;
202             thetaTemp=linspace(0,1,numPoints);
```

```
203
204         %make cells of the solutions;
205         xGrid=cell(numX,1);
206         fGrid=zeros(numPoints,1);
207         x0=mpsaEx{i}.lb;
208
209         options = optimoptions('fmincon',...
210             'Algorithm','sqp','Display','Off');
211         for idx=1:length(thetaTemp)
212             A=mpsaEx{i}.A;
213             b=mpsaEx{i}.b+mpsaEx{i}.E*thetaTemp(idx);
214             obj=@(x) mpsaEx{i}.objective(x,thetaTemp(idx));
215             const=@(x) mpsaEx{i}.constraints(x,thetaTemp(idx));
216             [tempX,fGrid(idx)]=fmincon(obj,x0,A,b,[],[],...
217                 mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
218             for idx2=1:numX
219                 xGrid{idx2}(idx)=tempX(idx2);
220             end
221         end %for
222
223         gurkeFig=figure;
224         plot(thetaTemp,fGrid,'o');
225         xlabel('\theta_1');
226         ylabel('f');
227         plotParametricFunction(fASM{i},CRASM{i});
228         title(['mpASM ' num2str(numX) ' Vars Optimal Value Function' ...
229             ' for Example' num2str(i)]);
230         filename=['ASM' num2str(numX) 'VarsVal' num2str(i) '.fig'];
231         saveas(gurkeFig,filename);
232     end%if
233 end
234
235
```

```
% %compute mp-SA
% for i=1:numExamples
%     %compute the example with the proper constraints
%     filename=['nonlcon' num2str(i) '.m'];
%     copyfile(filename, 'nonlcon.m');
%     mpsaEx{i}.constraints=@(x,theta) nonlcon(theta(1),x);
%
%     %triangulate Theta, take one simplex, and use an approximation for u0.
%     tempSimplex=Theta.triangulate.V;
%     vertices=tempSimplex;
%     %tempU=cell(length(vertices),1);
%
%     uStart=[];
%     %at each vertex, compute the solution and get the dual variable;
%     x0=mpsaEx{i}.lb;
%     A=mpsaEx{i}.A;
%     options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
%     for idx=1:length(vertices)
%         b=mpsaEx{i}.b+mpsaEx{i}.E*vertices(idx,:)';
%         obj=@(x) mpsaEx{i}.objective(x,vertices(idx,:));
%         const=@(x) mpsaEx{i}.constraints(x,vertices(idx,:));
%         [~,~,~,~,tempU,~,~]=fmincon(obj,x0,A,b,[],[],...
%             mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
%         uStart=[uStart;tempU.ineqnonlin'];
%     end %for
%
%     %when working with this in the future, use M*uInterp=uStart
%     %compute the linear interpolation for u0
%     M=[vertices, ones(length(vertices),1)];
%     Minv=M^-1;
%     %transpose so that we have each row being the interpolation
%     uInterp= (Minv*uStart)';
%     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%       u0{1}=uInterp;
%
%       %run mpSA
%       [xSA{i},uSA{i},vSA{i},fSA{i},CRSA{i},solnInfoSA{i}]=mpSA(u0,v0,0.01,...
%       Theta,mpsaEx{i},maxIter,cap);
%
%       %after computing, draw the graphs of the functions that have poor error
%       %values
%       if(1)
%           %make a grid pattern of the actual solution function
%           numPoints=21;
%           thetaTemp=linspace(0,1,numPoints);
%
%           %make cells of the solutions;
%           xGrid=cell(numX,1);
%           uGrid=cell(numNonlinConstraints,1);
%           fGrid=zeros(numPoints,1);
%           x0=mpsaEx{i}.lb;
%
%           options = optimoptions('fmincon',...
%               'Algorithm','sqp','Display','Off');
%           for idx=1:length(thetaTemp)
%               A=mpsaEx{i}.A;
%               b=mpsaEx{i}.b+mpsaEx{i}.E*thetaTemp(idx);
%               obj=@(x) mpsaEx{i}.objective(x,thetaTemp(idx));
%               const=@(x) mpsaEx{i}.constraints(x,thetaTemp(idx));
%               [tempX,fGrid(idx)]=fmincon(obj,x0,A,b,[],[],...
%                   mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
%               for idx2=1:numX
%                   xGrid{idx2}(idx)=tempX(idx2);
%               end
%               %define u here, but later
%           end %for
```

```
302 %
303 %           gurkeFig=figure;
304 %           plot(thetaTemp,fGrid,'o');
305 %           xlabel('\theta_1');
306 %           ylabel('f');
307 %           plotParametricFunction(fSA{i},CRSA{i});
308 %           title(['mpSA ' num2str(numX) ' Vars Optimal Value Function' ...
309 %                ' for Example' num2str(i)]);
310 %           filename=['SA' num2str(numX) 'VarsVal' num2str(i) '.fig'];
311 %           saveas(gurkeFig,filename);
312 %       end%if
313 % end
314
315 %collect the data
316 errorASM=zeros(1,numExamples);
317 xErrorASM=zeros(1,numExamples);
318 timeASM=zeros(1,numExamples);
319
320 for i=1:numExamples
321     errorASM(i)=solnInfoASM{i}.error;
322     xErrorASM(i)=solnInfoASM{i}.xError;
323     timeASM(i)=solnInfoASM{i}.time;
324 end
325
326 %for two variables:
327 % errorASM(6)=[];
328 % xErrorASM(6)=[];
329 % timeASM(6)=[];
330
331 %for three variables:
332 % errorASM(3)=[];
333 % xErrorASM(3)=[];
334 % timeASM(3)=[];
```

```
335
336 %for four variables:
337 % errorASM() =[];
338 % xErrorASM() =[];
339 % timeASM() =[];
340
341 %for five variables:
342 % errorASM() =[];
343 % xErrorASM() =[];
344 % timeASM() =[];
345
346
347 errorStats=struct;
348 errorStats.meanVal=mean(errorASM);
349 errorStats.medianVal=median(errorASM);
350 errorStats.stdDevVal=std(errorASM);
351
352 xErrStats=struct;
353 xErrStats.mean=mean(xErrorASM);
354 xErrStats.median=median(xErrorASM);
355 xErrStats.stdDev=std(xErrorASM);
356
357 timeStats=struct;
358 timeStats.mean=mean(timeASM);
359 timeStats.median=median(timeASM);
360 timeStats.stdDev=std(timeASM);
```

```
1 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      Matlab codes\mp ASM algorithm\mpASM')
2 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      MPT Practice\mpSAPractice')
3
4 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      MPT Practice\mpSAPractice\randFuncs')
```

```
5
6  %Random Example (plots)
7  %three variables
8  % numX=2;
9  % numX=3;
10 % numX=4;
11 numX=5;
12 % xSize=[1,numX];      %row form;
13
14 %MUST BE COLUMN FORM RIGHT NOW
15 xSize=[numX,1];       %col form;
16
17 x=sym('x',xSize);
18 assume(x, 'real');
19
20 %two parameters
21 syms theta1 theta2;
22 assume(theta1>=0 & theta1<=1);
23 assume(theta2>=0 & theta2<=1);
24
25 %ten examples, with 1 nonlinear and 2 linear constraints each
26 % numExamples=71;  %24 for 2 vars; , going from 0.1 to 0.9
27 % numExamples=43;  %43 for 3 vars; , going from 0.1 to 0.9
28 % numExamples=37;  %37 for 4 vars, going from 0.2 to 0.6
29 numExamples=29;  %29 for 5 vars, going from 0.2 to 0.6
30 numNonlinConstraints=1;
31 numLinCon=2;
32
33 %the parameter space for 4 and 5 variables
34 theta1Lower=0.2;
35 theta2Lower=0.2;
36 theta1Upper=0.6;
37 theta2Upper=0.6;
```

```
38
39 %the parameter space for 2 and 3 variables
40 % theta1Lower=0.1;
41 % theta2Lower=0.1;
42 % theta1Upper=0.9;
43 % theta2Upper=0.9;
44 thetaLower=[theta1Lower,theta2Lower];
45 thetaUpper=[theta1Upper,theta2Upper];
46 Theta = Polyhedron('lb',thetaLower,'ub',thetaUpper);
47
48 %depth for mp-ASM
49 cap=5;
50
51 %number of dual updates for mp-SA
52 maxIter=10;
53
54 %initial guess for dual variables: v does not exist
55 %Note u0={zeros(1,(1+numParams))};
56 u0={zeros(1,3)};
57 v0={};
58
59 %set the precision of the symbols
60 digits(5);
61
62 %set the seed so that there is some semblance of replicability
63 rng default;
64 seedNum=8+numX;
65 rng(seedNum);
66
67 %set the guaranteed feasible point
68 % bSelect=ones(xSize);       %if x is a row
69 bSelect=ones(fliplr(xSize));          %if x is a col
70
```

```
71  %Set the linear constraint Ax + E*theta<=b
72  Asize=[numLinCon,numX];
73  Esize=[numLinCon,2];    %2 being for theta1 and theta2
74  cSize=[1,numLinCon];
75
76  %matrices to help ensure feasibility for the linear constraints
77  thetaLowerRep=repmat(thetaLower,numLinCon,1);
78  thetaUpperRep=repmat(thetaUpper,numLinCon,1);
79
80  %the theta-picking matrix for the linear matrix
81  RaSize=[numLinCon,numLinCon];
82  RcSize=[numLinCon,numLinCon];
83
84  %the matrices for generating the objective function and the quadratic
85  %constraints
86  %Note Q=P'P
87  Psize=[numX,numX];
88  RpSize=Psize;
89  qSize=[1,numX];
90
91  cQuadSize=[1,1];
92  RcQuadSize=[1,1];
93
94  mpsaEx=cell(1,numExamples);
95  mpsaExTemp=struct;
96
97  %solution fields
98  xSA=cell(1,numExamples);
99  uSA=cell(1,numExamples);
100 vSA=cell(1,numExamples);
101 fSA=cell(1,numExamples);
102 CRSA=cell(1,numExamples);
103 solnInfoSA=cell(1,numExamples);
```

```
104
105 xASM=cell(1,numExamples);
106 fASM=cell(1,numExamples);
107 CRASM=cell(1,numExamples);
108 solnInfoASM=cell(1,numExamples);
109
110 %the constraints
111 gurke=cell(numExamples,numNonlinConstraints);
112 constraintsList=cell(numExamples,numNonlinConstraints);
113
114 %the objectives
115 objList=cell(numExamples,1);
116
117 for i=1:numExamples
118     disp(['Creating Example ' num2str(i) ' of ' num2str(numExamples)]);
119     mpsaExTemp.A=floor(randMat(Asize,0,1)*1000)/1000;
120
121     thetaSelect=rand(Esize);
122     mpsaExTemp.E=floor(randMat(Esize)*1000)/1000;
123     mpsaExTemp.E(thetaSelect<0.5)=0;
124
125     %we need that b gives feasibility for ALL theta, so it needs to
126     %incorporate E*theta in some way; use the lower bounds when E(i,j) is
127     %negative, and use the upper bounds when E(i,j) is positive
128     thetaAdjust=zeros(Esize);
129
130     thetaAdjust(mpsaExTemp.E<0)=thetaLowerRep(mpsaExTemp.E<0);
131     thetaAdjust(mpsaExTemp.E>=0)=thetaUpperRep(mpsaExTemp.E>=0);
132     mpsaExTemp.b=mpsaExTemp.A*bSelect'+ dot(mpsaExTemp.E,thetaAdjust,2);
133     mpsaExTemp.Aeq = [];
134     mpsaExTemp.Eeq = [];
135     mpsaExTemp.beq = [];
136
```

```
137     %unrelaxed nonlinear constraints
138     mpsaExTemp.baseConstraints=[];
139
140     %make the objective function: f(x;theta)=x^TQ(theta)x+q^T(theta)x
141     %Q=P'(theta)*P(theta)
142     %P(theta)=P1+P2*Rp*theta
143     P1=floor(randMat(Psize,0,1)*1000)/1000;
144     P2=floor(randMat(Psize,0,1)*1000)/1000;
145
146     Rp=floor(rand01(RpSize,0.5)*1000)/1000;
147
148     %q'(theta)*x
149     q1=floor(randMat(qSize,0,1)*1000)/1000;
150     q2=floor(randMat(qSize,0,1)*1000)/1000;
151     Rq=floor(rand01(RpSize,0.5)*1000)/1000;
152
153     P=vpa(P1+P2*Rp*theta1);
154     Q=expand(P'*P);
155     q=vpa(q1+q2*Rq*theta1);
156
157     %replace theta1^2 with theta2
158     temp=char(Q);
159     temp=erase(temp,'matrix(');
160     temp=erase(temp,')');
161     temp=strrep(temp,'theta1^2','theta2');
162     temp=str2sym(temp);
163     temp=reshape(temp,numX,numX);
164     Q=temp;
165
166 %       obj=x*Q*x'+q*x';      %if x is a row
167     obj=vpa(x'*Q*x+q*x);      %if x is a col
168     objList{i}=vpa(x'*Q*x+q*x);
169     obj=matlabFunction(obj,'Vars',{theta1,theta2,x});
```

```
170
171     %objective function; inner variables must be different than syms
172     mpsaExTemp.objective=@(varX,varTheta) ...
173     obj(varTheta(1),varTheta(2),varX);
174
175     for j=1:numNonlinConstraints
176         %Q=P'(theta)*P(theta)
177         %P(theta)=P1+P2*Rp*theta
178         P1=floor(randMat(Psize,0,1)*1000)/1000;
179         P2=floor(randMat(Psize,0,1)*1000)/1000;
180         Rp=floor(rand01(RpSize,0.5)*1000)/1000;
181
182         %q'(theta)*x
183         q1=floor(randMat(qSize,0,1)*1000)/1000;
184         q2=floor(randMat(qSize,0,1)*1000)/1000;
185         Rq=floor(rand01(RpSize,0.5)*1000)/1000;
186
187         %c(theta)
188         cQuad1=floor(randMat(cQuadSize,0,1)*1000)/1000;
189         cQuad2=floor(randMat(cQuadSize,0,1)*1000)/1000;
190         RcQuad=floor(rand01(cQuadSize,0.5)*1000)/1000;
191
192         P=vpa(P1+P2*Rp*theta1);
193         Q=expand(P'*P);
194         q=vpa(q1+q2*Rq*theta1);
195         cQuad=vpa(cQuad1+cQuad2*RcQuad*theta1);
196
197         %replace theta1^2 with theta2
198         temp=char(Q);
199         temp=erase(temp,'matrix(');
200         temp=erase(temp,')');
201         temp=strrep(temp,'theta1^2','theta2');
202         temp=str2sym(temp);
```

```
203             temp=reshape(temp,numX,numX);
204             Q=temp;
205
206             %b=bSelect'*Q(theta)*bSelect+q'(theta)*bSelect+cQuad(theta)
207             b=bSelect*Q*bSelect'+q*bSelect'+cQuad;
208
209             %generate the nonlinear constraints for mp-SA
210 %               gurke{j}=x*Q*x'+q*x'+cQuad-b;          %if x is a row
211             gurke{i,j}=vpa(x'*Q*x+q*x+cQuad-b);      %if x is a col
212             constraintsList{i,j}=gurke{i,j};
213             gurke{i,j}=matlabFunction(gurke{i,j},'Vars',{theta1,theta2,x});
214             %constraints to be relaxed; inner variables must be different than
215             %syms
216             mpsaExTemp.relConIneq{j}=@(varX,varTheta) ...
217                 gurke{i,j}(varTheta(1),varTheta(2),varX);
218         end
219         %create the nonlcon constraint for mp-ASM
220         if numNonlinConstraints>0
221             makeConstraints(gurke(i),true,numX,i);
222         end
223
224         mpsaExTemp.relConEq=cell(0);
225
226         %lower and upper bounds for x
227         %WARNING: need to be columns for mpASM
228 %       mpsaEx.lb = zeros(2,1);
229 %       mpsaEx.ub = inf*ones(2,1);
230 %       boundDim=fliplr(xSize);
231         boundDim=xSize;
232         mpsaExTemp.lb = zeros(boundDim);
233 %       mpsaExTemp.lb = -5*ones(boundDim);
234         mpsaExTemp.ub = 10^5*ones(boundDim);
235
```

```
236     mpsaEx{i}=mpsaExTemp;
237 end
238
239 %compute mp-ASM
240 for i=1:numExamples
241     %compute the example with the proper constraints
242     disp(['Solving Example ' num2str(i) ' of ' num2str(numExamples)]);
243     if numNonlinConstraints>0
244         filename=['nonlcon' num2str(numX) 'VarsEx' num2str(i) '.m'];
245         copyfile(filename, 'nonlcon.m');
246         mpsaEx{i}.constraints=@(x,theta) nonlcon(theta(1),theta(2),x);
247     else
248         mpsaEx{i}.constraints=[];
249     end
250
251     %run the same example to see how straight mpASM compares
252     [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
253         mpASM_preLEM(0.01,Theta,mpsaEx{i},cap);
254 end
255
256 %store how far the example stretches: 0.5 error is big for an example going
257 %from 1 to 3, but small for an example from 100 to 300
258 fRanges=zeros(1,numExamples);
259 %display the examples
260 for i=1:numExamples
261     %after computing, draw the graphs of the functions that have poor error
262     %values
263     disp(['Plotting Example ' num2str(i) ' of ' num2str(numExamples)]);
264     if(1)
265         %make a grid pattern of the actual solution function
266         numPoints=21;
267         thetaTemp=zeros(2,numPoints);
268         thetaTemp(1,:)=linspace(theta1Lower,theta1Upper,numPoints);
```

```
269         thetaTemp(2,:)=linspace(theta2Lower,theta2Upper,numPoints);
270
271         %make cells of the solutions; they're square for the surf command
272         xGrid=cell(numX,1);
273         for idx=1:numX
274             xGrid{idx}=zeros(numPoints,numPoints);
275         end
276         uGrid=cell(numNonlinConstraints,1);
277         for idx=1:numNonlinConstraints
278             uGrid{idx}=zeros(numPoints,numPoints);
279         end
280
281         fGrid=zeros(numPoints,numPoints);
282         x0=mpsaEx{i}.lb;
283
284         options = optimoptions('fmincon',...
285             'Algorithm','sqp','Display','Off');
286         for idx1=1:length(thetaTemp)
287             for idx2=1:length(thetaTemp)
288                 A=mpsaEx{i}.A;
289                 b=mpsaEx{i}.b+mpsaEx{i}.E*[thetaTemp(1,idx1);...
290                     thetaTemp(2,idx2)];
291                 objTemp=@(x) mpsaEx{i}.objective(x,[thetaTemp(1,idx1),...
292                     thetaTemp(2,idx2)]);
293             if numNonlinConstraints > 0
294                 const=@(x) mpsaEx{i}.constraints(x,[thetaTemp(1,idx1),...
295                     thetaTemp(2,idx2)]);
296             else
297                 const=[];
298             end
299
300                 %hth column, fill the rows
301                 [temp,fGrid(idx2,idx1),~,~,tempDual,~,~]=fmincon(objTemp,...
```

```
302                         x0,A,b,[],[],mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
303                     for idx=1:numX
304                         xGrid{idx}(idx2,idx1)=temp(idx);
305                     end
306                     for idx=1:numNonlinConstraints
307                         uGrid{idx}(idx2,idx1)=tempDual.ineqnonlin(idx);
308                     end
309
310                 end %for
311             end
312             [thetaX, thetaY]=meshgrid(thetaTemp(1,:),thetaTemp(2,:));
313             gurkeFig=figure;
314             surf(thetaX,thetaY,fGrid);
315             xlabel('\theta_1');
316             ylabel('\theta_2');
317             plotParametricFunction(fASM{i},CRASM{i});
318             title(['mpASM ' num2str(numX) ' Vars Optimal Value Function' ...
319                 ' for Example' num2str(i)]);
320
321             fRanges(i)=abs(max(fGrid(:))-min(fGrid(:)));
322         end%if
323 end
324
325 % %compute mp-SA
326 % for i=1:numExamples
327 %     %triangulate Theta, take one simplex, and use an approximation for u0.
328 %     tempSimplex=Theta.triangulate.V;
329 %     vertices=tempSimplex;
330 %     %tempU=cell(length(vertices),1);
331 %
332 %     uStart=[];
333 %     %at each vertex, compute the solution and get the dual variable;
334 %     x0=mpsaEx{i}.lb;
```

```
%     A=mpsaEx{i}.A;
%     options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
%     for idx=1:length(vertices)
%         b=mpsaEx{i}.b+mpsaEx{i}.E*vertices(idx,:)';
%         obj=@(x) mpsaEx{i}.objective(x,vertices(idx,:));
%         const=@(x) mpsaEx{i}.constraints(x,vertices(idx,:));
%         [~,~,~,~,tempU,~,~]=fmincon(obj,x0,A,b,[],[],...
%             mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
%         uStart=[uStart;tempU.ineqnonlin'];
%     end %for
%
%     %compute the linear interpolation for u0
%     M=[vertices, ones(length(vertices),1)];
%     Minv=M^-1;
%     %transpose so that we have each row being the interpolation
%     uInterp= (Minv*uStart)';
%     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     u0{1}=uInterp;
%
%     %run mpSA
%     [xSA{i},uSA{i},vSA{i},fSA{i},CRSA{i},solnInfoSA{i}]=mpSA(u0,v0,0.01,...
%     Theta,mpsaEx{i},maxIter,cap);
%
%     %after computing, draw the graphs of the functions that have poor error
%     %values
%     if(1)
%         %make a grid pattern of the actual solution function
%         numPoints=21;
%         thetaTemp=zeros(2,numPoints);
%         thetaTemp(1,:)=linspace(theta1Lower,theta1Upper,numPoints);
%         thetaTemp(2,:)=linspace(theta2Lower,theta2Upper,numPoints);
%
%         %make cells of the solutions; they're square for the surf command
```

```
368 %           xGrid=cell(numX,1);
369 %           for idx=1:numX
370 %               xGrid{idx}=zeros(numPoints,numPoints);
371 %           end
372 %           uGrid=cell(numNonlinConstraints,1);
373 %           for idx=1:numNonlinConstraints
374 %               uGrid{idx}=zeros(numPoints,numPoints);
375 %           end
376 %
377 %           fGrid=zeros(numPoints,numPoints);
378 %           x0=mpsaEx{i}.lb;
379 %
380 %           options = optimoptions('fmincon',...
381 %               'Algorithm','sqp','Display','Off');
382 %           for idx1=1:length(thetaTemp)
383 %               for idx2=1:length(thetaTemp)
384 %                   A=mpsaEx{i}.A;
385 %                   %WARNING: make this work right; check number of parameters
386 %                   b=mpsaEx{i}.b+mpsaEx{i}.E*[thetaTemp(1,idx1);...
387 %                       thetaTemp(2,idx2)];
388 %                   objTemp=@(x) mpsaEx{i}.objective(x,[thetaTemp(1,idx1),...
389 %                       thetaTemp(2,idx2)]);
390 %                   const=@(x) mpsaEx{i}.constraints(x,[thetaTemp(1,idx1),...
391 %                       thetaTemp(2,idx2)]);
392 %
393 %
394 %                   %hth column, fill the rows
395 %                   [temp,fGrid(idx2,idx1),~,~,tempDual,~,~]=fmincon(objTemp,...
396 %                       x0,A,b,[],[],mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
397 %                   for idx=1:numX
398 %                       xGrid{idx}(idx2,idx1)=temp(idx);
399 %                   end
400 %                   for idx=1:numNonlinConstraints
```

```
401 %                         uGrid{idx}(idx2,idx1)=tempDual.ineqnonlin(idx);
402 %                     end
403 %                 end %for
404 %             end
405 %             [thetaX, thetaY]=meshgrid(thetaTemp(1,:),thetaTemp(2,:));
406 %
407 %             gurkeFig=figure;
408 %             surf(thetaX,thetaY,fGrid);
409 %             xlabel('\theta_1');
410 %             ylabel('\theta_2');
411 %             plotParametricFunction(fSA{i},CRSA{i});
412 %             title(['mpSA ' num2str(numX) ' Vars Optimal Value Function' ...
413 %                 ' for Example' num2str(i)]);
414 %             filename=['SA' num2str(numX) 'VarsVal' num2str(i) '.fig'];
415 %             filename=fullfile('bigParamSpace',filename);
416 %             saveas(gurkeFig,filename);
417 %
418 %             gurkeFig=figure;
419 %             surf(thetaX,thetaY,fGrid);
420 %             xlabel('\theta_1');
421 %             ylabel('\theta_2');
422 %             plotParametricFunction(fASM{i},CRASM{i});
423 %             title(['mpASM ' num2str(numX) ' Vars Optimal Value Function' ...
424 %                 ' for Example' num2str(i)]);
425 %             filename=['ASM' num2str(numX) 'VarsVal' num2str(i) '.fig'];
426 %             filename=fullfile('bigParamSpace',filename);
427 %             saveas(gurkeFig,filename);
428 %         end%if
429 % end
430
431 %collect the data
432 rawErrASM=zeros(1,numExamples);
433 errorASM=zeros(1,numExamples);
```

```
434 xErrorASM=zeros(1,numExamples);
435 timeASM=zeros(1,numExamples);
436 numPolys=zeros(1,numExamples);
437 avgErr=zeros(1,numExamples);
438 avgScaledErr=zeros(1,numExamples);
439
440 for i=1:numExamples
441     rawErrASM(i)=solnInfoASM{i}.error;
442     errorASM(i)=rawErrASM(i)/fRanges(i);
443     xErrorASM(i)=solnInfoASM{i}.xError;
444     timeASM(i)=solnInfoASM{i}.time;
445     numPolys(i)=length(CRASM{i});
446     avgErr(i)=rawErrASM(i)/numPolys(i);
447     avgScaledErr(i)=errorASM(i)/numPolys(i);
448 end
449
450 %for two variables: omit 1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70
451 %keep:71, 62, 56, 43, 38, 30, 27, 5, 4, 2,
452 % rawErrASM([1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70])=[];
453 % errorASM([1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70])=[];
454 % xErrorASM([1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70])=[];
455 % timeASM([1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70])=[];
456 % numPolys([1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70])=[];
457 % avgErr([1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70])=[];
458 % avgScaledErr([1,3,6:26,28:29,31:37,39:42,44:55,57:61,63:70])=[];
459
460 %for three variables: omit 1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42
461 %keep: 43, 38, 31, 25, 24, 19, 16, 13, 12, 4,
462 % rawErrASM([1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42])=[];
463 % errorASM([1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42])=[];
464 % xErrorASM([1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42])=[];
465 % timeASM([1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42])=[];
466 % numPolys([1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42])=[];
```

```
467 % avgErr([1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42])=[];
468 % avgScaledErr([1:3,5:11,14:15,17:19,20:23,26:30,32:37,39:42])=[];
469
470 %for four variables: omit 2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36
471 %keep: 37, 35, 24, 20, 16, 15, 12, 8, 3, 1
472 % rawErrASM([2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36])=[];
473 % errorASM([2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36])=[];
474 % xErrorASM([2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36])=[];
475 % timeASM([2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36])=[];
476 % numPolys([2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36])=[];
477 % avgErr([2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36])=[];
478 % avgScaledErr([2, 4:7, 9:11, 13:14, 17:19, 21:23, 25:34, 36])=[];
479
480 %for five variables: omit 1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28
481 %keep: 4, 7, 9, 11, 19:20, 24, 25, 27, 29
482 rawErrASM([1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28])=[];
483 errorASM([1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28])=[];
484 xErrorASM([1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28])=[];
485 timeASM([1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28])=[];
486 numPolys([1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28])=[];
487 avgErr([1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28])=[];
488 avgScaledErr([1:3, 5:6, 8, 10, 12:18, 21:23, 26, 28])=[];
489
490 errorStats=struct;
491 errorStats.meanVal=mean(errorASM);
492 errorStats.medianVal=median(errorASM);
493 errorStats.stdDevVal=std(errorASM);
494
495 xErrStats=struct;
496 xErrStats.mean=mean(xErrorASM);
497 xErrStats.median=median(xErrorASM);
498 xErrStats.stdDev=std(xErrorASM);
499
```

```
timeStats=struct;
timeStats.mean=mean(timeASM);
timeStats.median=median(timeASM);
timeStats.stdDev=std(timeASM);

polyStats=struct;
polyStats.mean=mean(numPolys);
polyStats.median=median(numPolys);
polyStats.stdDev=std(numPolys);

avgErrStats=struct;
avgErrStats.mean=mean(avgErr);
avgErrStats.median=median(avgErr);
avgErrStats.stdDev=std(avgErr);

avgScaledErrStats=struct;
avgScaledErrStats.mean=mean(avgScaledErr);
avgScaledErrStats.median=median(avgScaledErr);
avgScaledErrStats.stdDev=std(avgScaledErr);
```

```
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm\mpASM')
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice')

addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice\randFuncs')

%Random Example (plots)
%three variables
% numX=2;
% numX=3;
% numX=4;
numX=5;
```

```
12 % xSize=[1,numX];        %row form;
13
14 %MUST BE COLUMN FORM RIGHT NOW
15 xSize=[numX,1];        %col form;
16
17 x=sym('x',xSize);
18 assume(x, 'real');
19
20 %two parameters
21 syms theta1 theta2;
22 assume(theta1>=0 & theta1<=1);
23 assume(theta2>=0 & theta2<=1);
24
25 %ten examples, with 1 nonlinear constraint each
26 % numExamples=36;  %36 for 2 vars; , going from 0.1 to 0.9
27 % numExamples=27;  %27 for 3 vars; , going from 0.1 to 0.9
28 % numExamples=26;  %26 for 4 vars, going from 0.2 to 0.6
29 numExamples=21;  %21 for 5 vars, going from 0.2 to 0.6
30 numNonlinConstraints=0;
31
32 %the parameter space for 4 and 5 variables
33 theta1Lower=0.2;
34 theta2Lower=0.2;
35 theta1Upper=0.6;
36 theta2Upper=0.6;
37
38 %the parameter space for 2 and 3 variables
39 % theta1Lower=0.1;
40 % theta2Lower=0.1;
41 % theta1Upper=0.9;
42 % theta2Upper=0.9;
43 thetaLower=[theta1Lower,theta2Lower];
44 thetaUpper=[theta1Upper,theta2Upper];
```

```
45 Theta = Polyhedron('lb',thetaLower,'ub',thetaUpper);
46
47 %depth for mp-ASM
48 cap=5;
49
50 %number of dual updates for mp-SA
51 maxIter=10;
52
53 %initial guess for dual variables: v does not exist
54 %Note u0={zeros(1,(1+numParams))};
55 u0={zeros(1,3)};
56 v0={};
57
58 %set the precision of the symbols
59 digits(5);
60
61 %set the seed so that there is some semblance of replicability
62 % seedNum=100; %20; doesn't work: 10
63 rng default;
64 seedNum=8+numX;
65 rng(seedNum);
66
67 %set the guaranteed feasible point
68 % bSelect=ones(xSize);       %if x is a row
69 bSelect=ones(fliplr(xSize));          %if x is a col
70
71 %Set the linear constraint Ax + E*theta<=b
72 numLinCon=3;
73 Asize=[numLinCon,numX];
74 Esize=[numLinCon,2];    %2 being for theta1 and theta2
75 cSize=[1,numLinCon];
76
77 %matrices to help ensure feasibility for the linear constraints
```

```
78  thetaLowerRep=repmat(thetaLower,numLinCon,1);
79  thetaUpperRep=repmat(thetaUpper,numLinCon,1);
80
81  %the theta-picking matrix for the linear matrix
82  RaSize=[numLinCon,numLinCon];
83  RcSize=[numLinCon,numLinCon];
84
85  %the matrices for generating the objective function and the quadratic
86  %constraints
87  %Note Q=P'P
88  Psize=[numX,numX];
89  RpSize=Psize;
90  qSize=[1,numX];
91
92  cQuadSize=[1,1];
93  RcQuadSize=[1,1];
94
95  mpsaEx=cell(1,numExamples);
96  mpsaExTemp=struct;
97
98  %solution fields
99  xSA=cell(1,numExamples);
100 uSA=cell(1,numExamples);
101 vSA=cell(1,numExamples);
102 fSA=cell(1,numExamples);
103 CRSA=cell(1,numExamples);
104 solnInfoSA=cell(1,numExamples);
105
106 xASM=cell(1,numExamples);
107 fASM=cell(1,numExamples);
108 CRASM=cell(1,numExamples);
109 solnInfoASM=cell(1,numExamples);
110
```

```
111 %the constraints
112 gurke=cell(numExamples,numNonlinConstraints);
113 constraintsList=cell(numExamples,numNonlinConstraints);
114
115 %the objectives
116 objList=cell(numExamples,1);
117
118 for i=1:numExamples
119     disp(['Creating Example ' num2str(i) ' of ' num2str(numExamples)]);
120     mpsaExTemp.A=floor(randMat(Asize,0,1)*1000)/1000;
121
122     thetaSelect=rand(Esize);
123     mpsaExTemp.E=floor(randMat(Esize)*1000)/1000;
124     mpsaExTemp.E(thetaSelect<0.5)=0;
125
126     %we need that b gives feasibility for ALL theta, so it needs to
127     %incorporate E*theta in some way; use the lower bounds when E(i,j) is
128     %negative, and use the upper bounds when E(i,j) is positive
129     thetaAdjust=zeros(Esize);
130
131     thetaAdjust(mpsaExTemp.E<0)=thetaLowerRep(mpsaExTemp.E<0);
132     thetaAdjust(mpsaExTemp.E>=0)=thetaUpperRep(mpsaExTemp.E>=0);
133     %see documentation of 'dot' for details
134     mpsaExTemp.b=mpsaExTemp.A*bSelect'+ dot(mpsaExTemp.E,thetaAdjust,2);
135     mpsaExTemp.Aeq = [];
136     mpsaExTemp.Eeq = [];
137     mpsaExTemp.beq = [];
138
139     %unrelaxed nonlinear constraints
140     mpsaExTemp.baseConstraints=[];
141
142     %make the objective function: f(x;theta)=x^TQ(theta)x+q^T(theta)x
143     %Q=P'(theta)*P(theta)
```

```
144     %P(theta)=P1+P2*Rp*theta
145     P1=floor(randMat(Psize,0,1)*1000)/1000;
146     P2=floor(randMat(Psize,0,1)*1000)/1000;
147
148     Rp=floor(rand01(RpSize,0.5)*1000)/1000;
149
150     %q'(theta)*x
151     q1=floor(randMat(qSize,0,1)*1000)/1000;
152     q2=floor(randMat(qSize,0,1)*1000)/1000;
153     Rq=floor(rand01(RpSize,0.5)*1000)/1000;
154
155     P=vpa(P1+P2*Rp*theta1);
156     Q=expand(P'*P);
157     q=vpa(q1+q2*Rq*theta1);
158
159     %replace theta1^2 with theta2
160     temp=char(Q);
161     temp=erase(temp,'matrix(');
162     temp=erase(temp,')');
163     temp=strrep(temp,'theta1^2','theta2');
164     temp=str2sym(temp);
165     temp=reshape(temp,numX,numX);
166     Q=temp;
167
168 %       obj=x*Q*x'+q*x';      %if x is a row
169       obj=vpa(x'*Q*x+q*x);      %if x is a col
170       objList{i}=vpa(x'*Q*x+q*x);
171       obj=matlabFunction(obj,'Vars',{theta1,theta2,x});
172
173       %objective function; inner variables must be different than syms
174       mpsaExTemp.objective=@(varX,varTheta) ...
175       obj(varTheta(1),varTheta(2),varX);
176
```

```
177     for j=1:numNonlinConstraints
178         %Q=P'(theta)*P(theta)
179         %P(theta)=P1+P2*Rp*theta
180         P1=floor(randMat(Psize,0,1)*1000)/1000;
181         P2=floor(randMat(Psize,0,1)*1000)/1000;
182         Rp=floor(rand01(RpSize,0.5)*1000)/1000;
183
184         %q'(theta)*x
185         q1=floor(randMat(qSize,0,1)*1000)/1000;
186         q2=floor(randMat(qSize,0,1)*1000)/1000;
187         Rq=floor(rand01(RpSize,0.5)*1000)/1000;
188
189         %c(theta)
190         cQuad1=floor(randMat(cQuadSize,0,1)*1000)/1000;
191         cQuad2=floor(randMat(cQuadSize,0,1)*1000)/1000;
192         RcQuad=floor(rand01(cQuadSize,0.5)*1000)/1000;
193
194         P=vpa(P1+P2*Rp*theta1);
195         Q=expand(P'*P);
196         q=vpa(q1+q2*Rq*theta1);
197         cQuad=vpa(cQuad1+cQuad2*RcQuad*theta1);
198
199         %replace theta1^2 with theta2
200         temp=char(Q);
201         temp=erase(temp,'matrix(');
202         temp=erase(temp,')');
203         temp=strrep(temp,'theta1^2','theta2');
204         temp=str2sym(temp);
205         temp=reshape(temp,numX,numX);
206         Q=temp;
207
208         %b=bSelect'*Q(theta)*bSelect+q'(theta)*bSelect+cQuad(theta)
209         b=bSelect*Q*bSelect'+q*bSelect'+cQuad;
```

```
210
211         %generate the nonlinear constraints for mp-SA
212 %           gurke{j}=x*Q*x'+q*x'+cQuad-b;         %if x is a row
213         gurke{i,j}=vpa(x'*Q*x+q*x+cQuad-b);     %if x is a col
214         constraintsList{i,j}=gurke{i,j};
215         gurke{i,j}=matlabFunction(gurke{i,j},'Vars',{theta1,theta2,x});
216         %constraints to be relaxed; inner variables must be different than
217         %syms
218         mpsaExTemp.relConIneq{j}=@(varX,varTheta) ...
219             gurke{i,j}(varTheta(1),varTheta(2),varX);
220     end
221     %create the nonlcon constraint for mp-ASM
222     if numNonlinConstraints>0
223         makeConstraints(gurke(i),true,numX,i);
224     end
225
226     mpsaExTemp.relConEq=cell(0);
227
228     %lower and upper bounds for x
229     %WARNING: need to be columns for mpASM
230 %     mpsaEx.lb = zeros(2,1);
231 %     mpsaEx.ub = inf*ones(2,1);
232 %     boundDim=fliplr(xSize);
233     boundDim=xSize;
234     mpsaExTemp.lb = zeros(boundDim);
235 %     mpsaExTemp.lb = -5*ones(boundDim);
236     mpsaExTemp.ub = 10^5*ones(boundDim);
237
238     mpsaEx{i}=mpsaExTemp;
239 end
240
241 %compute mp-ASM
242 for i=1:numExamples
```

```
243     %compute the example with the proper constraints
244     disp(['Solving Example ' num2str(i) ' of ' num2str(numExamples)]);
245     if numNonlinConstraints>0
246         filename=['nonlcon' num2str(numX) 'VarsEx' num2str(i) '.m'];
247         copyfile(filename, 'nonlcon.m');
248         mpsaEx{i}.constraints=@(x,theta) nonlcon(theta(1),theta(2),x);
249     else
250         mpsaEx{i}.constraints=[];
251     end
252
253     %run the same example to see how straight mpASM compares
254     [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
255         mpASM_preLEM(0.01,Theta,mpsaEx{i},cap);
256 end
257
258 %store how far the example stretches: 0.5 error is big for an example going
259 %from 1 to 3, but small for an example from 100 to 300
260 fRanges=zeros(1,numExamples);
261 %display the examples
262 for i=1:numExamples
263     %after computing, draw the graphs of the functions that have poor error
264     %values
265     disp(['Plotting Example ' num2str(i) ' of ' num2str(numExamples)]);
266     if(1)
267         %make a grid pattern of the actual solution function
268         numPoints=21;
269         thetaTemp=zeros(2,numPoints);
270         thetaTemp(1,:)=linspace(theta1Lower,theta1Upper,numPoints);
271         thetaTemp(2,:)=linspace(theta2Lower,theta2Upper,numPoints);
272
273         %make cells of the solutions; they're square for the surf command
274         xGrid=cell(numX,1);
275         for idx=1:numX
```

```
276             xGrid{idx}=zeros(numPoints,numPoints);
277         end
278         uGrid=cell(numNonlinConstraints,1);
279         for idx=1:numNonlinConstraints
280             uGrid{idx}=zeros(numPoints,numPoints);
281         end
282
283         fGrid=zeros(numPoints,numPoints);
284         x0=mpsaEx{i}.lb;
285
286         options = optimoptions('fmincon',...
287             'Algorithm','sqp','Display','Off');
288         for idx1=1:length(thetaTemp)
289             for idx2=1:length(thetaTemp)
290                 A=mpsaEx{i}.A;
291                 b=mpsaEx{i}.b+mpsaEx{i}.E*[thetaTemp(1,idx1);...
292                     thetaTemp(2,idx2)];
293                 objTemp=@(x) mpsaEx{i}.objective(x,[thetaTemp(1,idx1),...
294                     thetaTemp(2,idx2)]);
295             if numNonlinConstraints > 0
296                 const=@(x) mpsaEx{i}.constraints(x,[thetaTemp(1,idx1),...
297                     thetaTemp(2,idx2)]);
298             else
299                 const=[];
300             end
301
302                 %hth column, fill the rows
303                 [temp,fGrid(idx2,idx1),~,~,tempDual,~,~]=fmincon(objTemp,...
304                     x0,A,b,[],[],mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
305                 for idx=1:numX
306                     xGrid{idx}(idx2,idx1)=temp(idx);
307                 end
308                 for idx=1:numNonlinConstraints
```

```
309                         uGrid{idx}(idx2,idx1)=tempDual.ineqnonlin(idx);
310                     end
311
312                 end %for
313             end
314             [thetaX, thetaY]=meshgrid(thetaTemp(1,:),thetaTemp(2,:));
315             gurkeFig=figure;
316             surf(thetaX,thetaY,fGrid);
317             xlabel('\theta_1');
318             ylabel('\theta_2');
319             plotParametricFunction(fASM{i},CRASM{i});
320             title(['mpASM ' num2str(numX) ' Vars Optimal Value Function' ...
321                 ' for Example' num2str(i)]);
322
323             fRanges(i)=abs(max(fGrid(:))-min(fGrid(:)));
324         end%if
325 end
326
327 % %compute mp-SA
328 % for i=1:numExamples
329 %     %triangulate Theta, take one simplex, and use an approximation for u0.
330 %     tempSimplex=Theta.triangulate.V;
331 %     vertices=tempSimplex;
332 %     %tempU=cell(length(vertices),1);
333 %
334 %     uStart=[];
335 %     %at each vertex, compute the solution and get the dual variable;
336 %     x0=mpsaEx{i}.lb;
337 %     A=mpsaEx{i}.A;
338 %     options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
339 %     for idx=1:length(vertices)
340 %         b=mpsaEx{i}.b+mpsaEx{i}.E*vertices(idx,:)';
341 %         obj=@(x) mpsaEx{i}.objective(x,vertices(idx,:));
```

```
%             const=@(x) mpsaEx{i}.constraints(x,vertices(idx,:));
%             [~,~,~,~,tempU,~,~]=fmincon(obj,x0,A,b,[],[],...
%                 mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
%             uStart=[uStart;tempU.ineqnonlin'];
%         end %for
%
%         %compute the linear interpolation for u0
%         M=[vertices, ones(length(vertices),1)];
%         Minv=M^-1;
%         %transpose so that we have each row being the interpolation
%         uInterp= (Minv*uStart)';
%         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%         u0{1}=uInterp;
%
%         %run mpSA
%         [xSA{i},uSA{i},vSA{i},fSA{i},CRSA{i},solnInfoSA{i}]=mpSA(u0,v0,0.01,...
%         Theta,mpsaEx{i},maxIter,cap);
%
%         %after computing, draw the graphs of the functions that have poor error
%         %values
% %       if(solnInfoSA{i}.meanErr1>0.01)
%         if(1)
%             %make a grid pattern of the actual solution function
%             numPoints=21;
%             thetaTemp=zeros(2,numPoints);
%             thetaTemp(1,:)=linspace(theta1Lower,theta1Upper,numPoints);
%             thetaTemp(2,:)=linspace(theta2Lower,theta2Upper,numPoints);
%
%             %make cells of the solutions; they're square for the surf command
%             xGrid=cell(numX,1);
%             for idx=1:numX
%                 xGrid{idx}=zeros(numPoints,numPoints);
%             end
```

```
375 %           uGrid=cell(numNonlinConstraints,1);
376 %           for idx=1:numNonlinConstraints
377 %               uGrid{idx}=zeros(numPoints,numPoints);
378 %           end
379 %
380 %           fGrid=zeros(numPoints,numPoints);
381 %           x0=mpsaEx{i}.lb;
382 %
383 %           options = optimoptions('fmincon',...
384 %               'Algorithm','sqp','Display','Off');
385 %           for idx1=1:length(thetaTemp)
386 %               for idx2=1:length(thetaTemp)
387 %                   A=mpsaEx{i}.A;
388 %                   %WARNING: make this work right; check number of parameters
389 %                   b=mpsaEx{i}.b+mpsaEx{i}.E*[thetaTemp(1,idx1);...
390 %                       thetaTemp(2,idx2)];
391 %                   objTemp=@(x) mpsaEx{i}.objective(x,[thetaTemp(1,idx1),...
392 %                       thetaTemp(2,idx2)]);
393 %                   const=@(x) mpsaEx{i}.constraints(x,[thetaTemp(1,idx1),...
394 %                       thetaTemp(2,idx2)]);
395 %
396 %                   %hth row, fill the columns
397 %           %           [temp,fGrid(idx1,idx2),~,~,tempDual,~,~]=fmincon(objTemp
    ,...
398 %           %               x0,A,b,[],[],mpsaEx.lb,mpsaEx.ub,const,options);
399 %
400 %                   %hth column, fill the rows
401 %                   [temp,fGrid(idx2,idx1),~,~,tempDual,~,~]=fmincon(objTemp,...
402 %                       x0,A,b,[],[],mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
403 %                   for idx=1:numX
404 %                       xGrid{idx}(idx2,idx1)=temp(idx);
405 %                   end
406 %                   for idx=1:numNonlinConstraints
```

```
407 %                         uGrid{idx}(idx2,idx1)=tempDual.ineqnonlin(idx);
408 %                     end
409 %                 end %for
410 %             end
411 %             [thetaX, thetaY]=meshgrid(thetaTemp(1,:),thetaTemp(2,:));
412 %
413 %             gurkeFig=figure;
414 %             surf(thetaX,thetaY,fGrid);
415 %             xlabel('\theta_1');
416 %             ylabel('\theta_2');
417 %             plotParametricFunction(fSA{i},CRSA{i});
418 %             title(['mpSA ' num2str(numX) ' Vars Optimal Value Function' ...
419 %                 ' for Example' num2str(i)]);
420 %             filename=['SA' num2str(numX) 'VarsVal' num2str(i) '.fig'];
421 %             filename=fullfile('bigParamSpace',filename);
422 %             saveas(gurkeFig,filename);
423 %
424 %             gurkeFig=figure;
425 %             surf(thetaX,thetaY,fGrid);
426 %             xlabel('\theta_1');
427 %             ylabel('\theta_2');
428 %             plotParametricFunction(fASM{i},CRASM{i});
429 %             title(['mpASM ' num2str(numX) ' Vars Optimal Value Function' ...
430 %                 ' for Example' num2str(i)]);
431 %             filename=['ASM' num2str(numX) 'VarsVal' num2str(i) '.fig'];
432 %             filename=fullfile('bigParamSpace',filename);
433 %             saveas(gurkeFig,filename);
434 %         end%if
435 % end
436
437 %collect the data
438 rawErrASM=zeros(1,numExamples);
439 errorASM=zeros(1,numExamples);
```

```
440 xErrorASM=zeros(1,numExamples);
441 timeASM=zeros(1,numExamples);
442 numPolys=zeros(1,numExamples);
443 avgErr=zeros(1,numExamples);
444 avgScaledErr=zeros(1,numExamples);
445
446 for i=1:numExamples
447     rawErrASM(i)=solnInfoASM{i}.error;
448     errorASM(i)=rawErrASM(i)/fRanges(i);
449     xErrorASM(i)=solnInfoASM{i}.xError;
450     timeASM(i)=solnInfoASM{i}.time;
451     numPolys(i)=length(CRASM{i});
452     avgErr(i)=rawErrASM(i)/numPolys(i);
453     avgScaledErr(i)=errorASM(i)/numPolys(i);
454 end
455
456 %for two variables: omit 1:3, 7:10, 12, 15:21, 23:24, 27:34
457 %keep: 4, 6, 11, 13, 14, 22, 25, 26, 35, 36
458 % rawErrASM([1:3, 7:10, 12, 15:21, 23:24, 27:34])=[];
459 % errorASM([1:3, 7:10, 12, 15:21, 23:24, 27:34])=[];
460 % xErrorASM([1:3, 7:10, 12, 15:21, 23:24, 27:34])=[];
461 % timeASM([1:3, 7:10, 12, 15:21, 23:24, 27:34])=[];
462 % numPolys([1:3, 7:10, 12, 15:21, 23:24, 27:34])=[];
463 % avgErr([1:3, 7:10, 12, 15:21, 23:24, 27:34])=[];
464 % avgScaledErr([1:3, 7:10, 12, 15:21, 23:24, 27:34])=[];
465
466 %for three variables: omit 1:3, 5, 7:11, 13:16, 20, 22, 26
467 %keep: 27, 25, 24, 23, 21, 19,18, 12, 6, 4,
468 % rawErrASM([1:3, 5, 7:11, 13:17, 20, 22, 26])=[];
469 % errorASM([1:3, 5, 7:11, 13:17, 20, 22, 26])=[];
470 % xErrorASM([1:3, 5, 7:11, 13:17, 20, 22, 26])=[];
471 % timeASM([1:3, 5, 7:11, 13:17, 20, 22, 26])=[];
472 % numPolys([1:3, 5, 7:11, 13:17, 20, 22, 26])=[];
```

```
473 % avgErr([1:3, 5, 7:11, 13:17, 20, 22, 26])=[];
474 % avgScaledErr([1:3, 5, 7:11, 13:17, 20, 22, 26])=[];
475
476 %for four variables: omit 2, 4:8, 13:18, 19, 22:25
477 %keep: 27, 26, 21, 20, 19, 12, 11, 10, 9, 3, 1
478 % rawErrASM([2, 4:8, 13:18, 19, 22:25])=[];
479 % errorASM([2, 4:8, 13:18, 19, 22:25])=[];
480 % xErrorASM([2, 4:8, 13:18, 19, 22:25])=[];
481 % timeASM([2, 4:8, 13:18, 19, 22:25])=[];
482 % numPolys([2, 4:8, 13:18, 19, 22:25])=[];
483 % avgErr([2, 4:8, 13:18, 19, 22:25])=[];
484 % avgScaledErr([2, 4:8, 13:18, 19, 22:25])=[];
485
486 %for five variables: omit 2:3, 7, 10:11, 13, 15:19
487 %keep: 21, 20, 14, 12, 9, 8, 6, 5, 4, 1
488 rawErrASM([2:3, 7, 10:11, 13, 15:19])=[];
489 errorASM([2:3, 7, 10:11, 13, 15:19])=[];
490 xErrorASM([2:3, 7, 10:11, 13, 15:19])=[];
491 timeASM([2:3, 7, 10:11, 13, 15:19])=[];
492 numPolys([2:3, 7, 10:11, 13, 15:19])=[];
493 avgErr([2:3, 7, 10:11, 13, 15:19])=[];
494 avgScaledErr([2:3, 7, 10:11, 13, 15:19])=[];
495
496 errorStats=struct;
497 errorStats.meanVal=mean(errorASM);
498 errorStats.medianVal=median(errorASM);
499 errorStats.stdDevVal=std(errorASM);
500
501 xErrStats=struct;
502 xErrStats.mean=mean(xErrorASM);
503 xErrStats.median=median(xErrorASM);
504 xErrStats.stdDev=std(xErrorASM);
505
```

```
timeStats=struct;
timeStats.mean=mean(timeASM);
timeStats.median=median(timeASM);
timeStats.stdDev=std(timeASM);

polyStats=struct;
polyStats.mean=mean(numPolys);
polyStats.median=median(numPolys);
polyStats.stdDev=std(numPolys);

avgErrStats=struct;
avgErrStats.mean=mean(avgErr);
avgErrStats.median=median(avgErr);
avgErrStats.stdDev=std(avgErr);

avgScaledErrStats=struct;
avgScaledErrStats.mean=mean(avgScaledErr);
avgScaledErrStats.median=median(avgScaledErr);
avgScaledErrStats.stdDev=std(avgScaledErr);
```

B.3 Elastic Net Example

The code for Section 2.5.1 is given below.

```
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm\mpASM')
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice')

%note that x4, x5, and x6 are just the absolute value variables
P=[1,2,1;1,3,2;1,2,2;1,4,5;1,3.5,2.5];

y=[0.5502;0.6225;0.5870;0.5689;0.7580];

```

```
c=y'*y;

A=[0  0  0  ones(1,3);
     1  0  0 -1  0  0;
     -1  0  0 -1  0  0;
     0  1  0  0 -1  0;
     0 -1  0  0 -1  0;
     0  0  1  0  0 -1;
     0  0 -1  0  0 -1];

b=zeros(7,1);

%x4, x5, and x6 are nonnegative
lb= [-50*ones(3,1); zeros(3,1)];
ub= inf*ones(6,1);

%initial guess
x0=lb;

%must receive column x;
%Q=P'P;      p=-P'y;      c=y'y;
f1=@(x) norm(P*x(1:3)-y,2)^2;          %||Px-b||^2

f2=@(x) norm(x(1:3),2)^2;
f3=@(x) norm(x(1:3),1);

%optimization problem structure
mpsaMOP = struct;
%parabolic cylinder objective function handle
mpsaMOP.objective = @(x,theta) f1(x);

%linear constraint matrices (inequality and equality)
%Ax<=b+E\theta
```

```
42 mpsaMOP.A = [];
43 mpsaMOP.b = [];
44 mpsaMOP.E = [];
45 mpsaMOP.Aeq = [];
46 mpsaMOP.Eeq = [];
47 mpsaMOP.beq = [];
48 lb=-50*ones(3,1);
49 ub=inf*ones(3,1);
50
51 %nonlinear constraints (for error computation, only)
52 mpsaMOP.constraints=@(x,epsilon) nonlconREC(x,epsilon,f2,f3);
53
54 %unrelaxed nonlinear constraints
55 mpsaMOP.baseConstraints=[];
56
57 %constraints to be relaxed;
58 mpsaMOP.relConIneq{1}=@(x,epsilon) epsilon(1)*f2(x)+(1-epsilon(1))*...
59     f3(x)-epsilon(2);
60 mpsaMOP.relConEq=cell(0);
61
62 %lower and upper bounds for x
63 mpsaMOP.lb = lb;
64 mpsaMOP.ub = ub;
65
66
67 %argmin(f1)
68 xMinf1=(P'*P)^-1*P'*y;
69
70 %note that eps2Max is acting as alpha
71 eps2Max=1;
72 eps3Max=0.6;
73 Epsilon = Polyhedron('lb',[0,0],'ub',[eps2Max,eps3Max]);
74 %note that Epsilon contains (alpha, epsilon)
```

```
75
76  [x01ASM,f01ASM,CR01ASM,solnInfo01ASM] = mpASM_preLEM(0.01,Epsilon,mpsaMOP);
77
78  %extracting new data points (2-7-2021)
79  % \lambda=0.955, \epsilon=0.087
80  % wahX=x01ASM{CR01ASM.contains(Polyhedron([0.955, 0.087]))};
81  % f01ASM{CR01ASM.contains(Polyhedron([0.955, 0.087]))};
82  % CR01ASM(CR01ASM.contains(Polyhedron([0.955, 0.087]))).H;
83  %
84  % wah1=f2(x01ASM{CR01ASM.contains(Polyhedron([0.955, 0.07]))}*[0.955;0.087;1])
        ;
85  % wah2=f3(x01ASM{CR01ASM.contains(Polyhedron([0.955, 0.07]))}*[0.955;0.087;1])
        ;
86  % wahRror=0.955*wah1+(1-0.955)*wah2;
87
88
89  clf('reset');
90  dotColor=[0.3010 0.7450 0.9330];
91  hold on;
92  plotParametricFunction(f01ASM,CR01ASM);
93  xlabel('\lambda');
94  ylabel('\epsilon');
95  zlabel('f_1');
96  title('min f_1 st \lambda\cdot f_2 + (1-\lambda)f_3\leq\epsilon;     mpASM');
97  hold off;
98
99
100 %make a grid pattern of the actual solution function, heavier closer to
101 %zero, sparser closer to {eps2Max,eps3Max}
102 numPoints=21;
103 lambda=zeros(2,numPoints);
104 lambda(1,:)=[logspace(-3,log10(eps2Max),numPoints)];
105 lambda(2,:)=[logspace(-3,log10(eps3Max),numPoints)];
```

```
106
107 fGrid=zeros(numPoints,numPoints);
108 x0=mpsaMOP.lb;
109
110 options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
111
112 for h=1:length(lambda)
113     for i=1:length(lambda)
114         A=mpsaMOP.A;
115         b=mpsaMOP.b;
116         obj=@(x) mpsaMOP.objective(x,[lambda(1,h),lambda(2,i)]);
117         const=@(x) mpsaMOP.constraints(x,[lambda(1,h),lambda(2,i)]);
118
119         %hth column, fill the rows
120         [temp,fGrid(i,h),~,~,tempDual,~,~]=fmincon(obj,...
121             x0,A,b,[],[],mpsaMOP.lb,mpsaMOP.ub,const,options);
122     end %for
123 end
124
125 [lambda1, lambda2]=meshgrid(lambda(1,:),lambda(2,:));
126
127 figure;
128 dotColor=[0.3010 0.7450 0.9330];
129 hold on;
130 surf(lambda1,lambda2,fGrid);
131 plotParametricFunction(f01ASM,CR01ASM);
132 xlabel('\lambda');
133 ylabel('\epsilon');
134 zlabel('f_1');
135 title('min f_1 st \lambda\cdot f_2 + (1-\lambda)f_3\leq\epsilon;     mpASM');
136
137 figure;
138 hold on;
```

```
139 surf(lambda1,lambda2,fGrid);
140 plotParametricFunction(f01,CR01);
141 xlabel('\alpha');
142 ylabel('\epsilon');
143 zlabel('f_1');
144 title('min f_1 st \alpha\cdot f_2 + (1-\alpha)f_3\leq\epsilon;     mpSA');
145 hold off;
146
147 %check
148 %f_2(x*(epsilon_2, epsilon_3))= epsilon_2:
149 %for each CR, try to compute the points \epsilon at which both f are active
150 options=optimoptions('fsolve','Display','off');
151 xParetoRegion=cell(0);
152 f1ParetoRegion=cell(0);
153 CRPareto=[];
154 epsPareto=[];
155 problemIndices=[];
156 gurke=1;
157 for i=1:length(CR01ASM)
158     %for each element of lambda, compute f_i(x(lambda))
159     systSolve=@(epsilon) [f2(x01ASM{i}*[epsilon,1]')-epsilon(1), f3(...
160                           x01ASM{i}*[epsilon,1]')-epsilon(2)];
161     for idx=1:1      %3 is the number of vertices in each CR
162         eps0=CR01ASM(i).V(idx,:);
163
164         [activeEps,~,exitFlag,~]=fsolve(systSolve,eps0,options);
165         res2=abs(f2(x01ASM{i}*[activeEps,1]')-activeEps(1));
166         res3=abs(f3(x01ASM{i}*[activeEps,1]')-activeEps(2));
167
168         %only keep the numbers if the constraint norms are close enough
169         tol=0.00001;
170
171         if res2<tol && res3<tol && activeEps(1)>0 && activeEps(2)>0
```

```
172             %the epsilons are a solution; now find where they are located
173             for j=1:length(CR01ASM)
174                 if CR01ASM(j).contains(activeEps')
175                     xParetoRegion{gurke,1}=x01ASM{j};
176                     f1ParetoRegion{gurke,1}=f01ASM{j};
177                     CRPareto=[CRPareto; CR01ASM(j)];
178                     epsPareto=[epsPareto; activeEps];
179                     gurke=gurke+1;
180                     break; %a CR containing this pareto point was found;
181                             %stop looking
182                 end
183             end
184         end
185     end %vertex for
186 end
187
188 gurke=gurke-1;
189 %note that there are x1,x2,x3 and x4, x5, x6 (the absolute value vars)
190 xPareto=zeros(gurke,3);
191 f1Pareto=zeros(gurke,1);
192 for i=1:gurke
193     xPareto(i,:)=xParetoRegion{i}*[epsPareto(i,:),1]';
194     f1Pareto(i)=f1ParetoRegion{i}*[epsPareto(i,:),1]';
195 end
196 figure;
197 plotParametricFunction(f1ParetoRegion,CRPareto);
198 plot3(epsPareto(:,1),epsPareto(:,2),f1Pareto,'o','MarkerSize',10,...
199     'MarkerFaceColor','g');
200 xlabel('\epsilon_2');
201 ylabel('\epsilon_3');
202 zlabel('f_1');
203
204 title(['min f_1 st \alpha\cdot f_2 + (1-\alpha)f_3\leq\epsilon;' ...
```

```
205     newline 'mpASM,    Pareto Dots   \alpha\in[0,' num2str(eps2Max)...
206     '], \epsilon\in[0,' num2str(eps3Max) ']']);
207 % %Pubudu's points
208 %  f2       f3        f1
209 %0.1953   0.6054    0.01164
210 %0.03622  0.2639    0.08381
211 % 0.0375   0.1937   0.1109
212 %0        0         1.933
213 plot3(0.1953,0.6054,0.01164, '.k','MarkerSize',75);
214 plot3(0.03622, 0.2639,0.08381, '.k','MarkerSize',75);
215 plot3( 0.0375, 0.1937, 0.1109 , '.k','MarkerSize',75);
216 plot3(0,0,1.933, '.k','MarkerSize',75);
217
218 %  X(f1)    Y(f2)    Z(f3)
219 %0.01164   0.1953   0.6054
220 %0.08381   0.03622  0.2639
221 %0.1109    0.0375   0.1937
222 %1.933     0        0
223
224 figure;
225 plotParametricFunction(xParetoRegion,CRPareto,[],[],1);
226 plot3(epsPareto(:,1),epsPareto(:,2),xPareto(:,1),'o','MarkerSize',10,...
227     'MarkerFaceColor','g');
228 xlabel('\epsilon_2');
229 ylabel('\epsilon_3');
230 zlabel('x_1');
231 title(['min f_1 st \alpha\cdot f_2 + (1-\alpha)f_3\leq\epsilon;' ...
232     newline 'mpASM,    Pareto Dots   \alpha\in[0,' num2str(eps2Max)...
233     '], \epsilon\in[0,' num2str(eps3Max) ']']);
234
235 figure;
236 plotParametricFunction(xParetoRegion,CRPareto,[],[],2);
237 plot3(epsPareto(:,1),epsPareto(:,2),xPareto(:,2),'o','MarkerSize',10,...
```

```
    'MarkerFaceColor','g');
xlabel('\alpha');
ylabel('\epsilon');
zlabel('x_2');
title(['min f_1 st \alpha\cdot f_2 + (1-\alpha)f_3\leq\epsilon;' ...
    newline 'mpASM,    Pareto Dots   \alpha\in[0,' num2str(eps2Max)...
    '], \epsilon\in[0,' num2str(eps3Max) ']']);

figure;
plotParametricFunction(xParetoRegion,CRPareto,[],[],3);
plot3(epsPareto(:,1),epsPareto(:,2),xPareto(:,3),'o','MarkerSize',10,...
    'MarkerFaceColor','g');
xlabel('\alpha');
ylabel('\epsilon');
zlabel('x_3');
title(['min f_1 st \alpha\cdot f_2 + (1-\alpha)f_3\leq\epsilon;' ...
    newline 'mpASM,    Pareto Dots   \alpha\in[0,' num2str(eps2Max)...
    '], \epsilon\in[0,' num2str(eps3Max) ']']);
```

```
%the nonlinear constraint for nonSqRedEpsCon
function [g, h] = nonlconREC(x,theta,func1,func2)
    %note that f3 is the 1-norm of the original x values
    g(1)=theta(1)*func1(x)+(1-theta(1))*func2(x)-theta(2);

    h=[];
end
```

```
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm\mpASM')
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice')

%note that x4, x5, and x6 are just the absolute value variables
P=[1,2,1;1,3,2;1,2,2;1,4,5;1,3.5,2.5];
```

```
6
7  y=[0.5502;0.6225;0.5870;0.5689;0.7580];
8
9  c=y'*y;
10
11 A=[0  0  0  ones(1,3);
12       1  0  0 -1  0  0;
13      -1  0  0 -1  0  0;
14       0  1  0  0 -1  0;
15       0 -1  0  0 -1  0;
16       0  0  1  0  0 -1;
17       0  0 -1  0  0 -1];
18
19 b=zeros(7,1);
20
21 %x4, x5, and x6 are nonnegative
22 lb= [-50*ones(3,1); zeros(3,1)];
23 ub= inf*ones(6,1);
24
25 %initial guess
26 x0=lb;
27
28 %must receive column x;
29 %Q=P'P;      p=-P'y;      c=y'y;
30 f1=@(x) norm(P*x(1:3)-y,2)^2;          %||Px-b||^2
31
32 f2=@(x) norm(x(1:3),2)^2;
33 f3=@(x) norm(x(1:3),1);
34
35 %optimization problem structure
36 mpsaMOP = struct;
37 %parabolic cylinder objective function handle
38 mpsaMOP.objective = @(x,theta) f1(x);
```

```
39
40 %linear constraint matrices (inequality and equality)
41 %Ax<=b+E\theta
42 mpsaMOP.A = A;
43 mpsaMOP.b = b;
44 mpsaMOP.E = [0,1; zeros(6,2)];
45 mpsaMOP.Aeq = [];
46 mpsaMOP.Eeq = [];
47 mpsaMOP.beq = [];
48
49 %nonlinear constraints (for error computation, only)
50 mpsaMOP.constraints=@(x,epsilon) nonlcon(x,epsilon,f2);
51
52 %unrelaxed nonlinear constraints
53 mpsaMOP.baseConstraints=[];
54
55 %constraints to be relaxed;
56 mpsaMOP.relConIneq{1}=@(x,epsilon) f2(x)-epsilon(1);
57 mpsaMOP.relConEq=cell(0);
58 % mpsaEx.relConEq{1}=@(x,theta) (x(1)-2)^2+(x(2)-2)^2-4-theta;
59 % mpsaEx.relConIneq=cell(0);
60
61 %lower and upper bounds for x
62 mpsaMOP.lb = lb;
63 mpsaMOP.ub = ub;
64
65 %pareto point
66 %argmin(f1)
67 xMinf1=(P'*P)^-1*P'*y;
68 eps2Max=f2(xMinf1);
69 eps3Max=f3(xMinf1);
70
71 Epsilon = Polyhedron('lb',[0,0],'ub',[eps2Max,eps3Max]);
```

```
%note that Epsilon contains (epsilon2, epsilon3)

%to compare how the solutions looked with different split depths; 5 was
%what we wanted
for split=5
[x01ASM,f01ASM,CR01ASM,solnInfo01ASM] = mpASM_preLEM(0.01,Epsilon,mpsaMOP,
    split);

%extracting data points (12-15-2020)
% x01ASM{CR01ASM.contains(Polyhedron([0.1188, 0.5061]))};
% f01ASM{CR01ASM.contains(Polyhedron([0.1188, 0.5061]))}*[0.1188;0.5061;1];
% CR01ASM(CR01ASM.contains(Polyhedron([0.1188, 0.5061]))).H

% f2(x01ASM{CR01ASM.contains(Polyhedron([0.1188, 0.5061]))}*[0.1188;0.5061;1])
% f3(x01ASM{CR01ASM.contains(Polyhedron([0.1188, 0.5061]))}*[0.1188;0.5061;1])

figure
%change the font to Times New Roman
set(0,'DefaultAxesFontName','Times New Roman');
set(0,'DefaultAxesFontSize',12);
set(0,'DefaultTextFontName','Times New Roman');
set(0,'DefaultTextFontSize',12);
set(0,'DefaultAxesFontWeight','normal');
set(0,'DefaultTextFontWeight','normal');
dotColor='black';
grey=[0.5 0.5 0.5];
hold on;
plotParametricFunction(f01ASM,CR01ASM);
xlabel('\boldmath$\epsilon_2$','Interpreter','latex');
ylabel('\boldmath$\epsilon_3$','Interpreter','latex');
zlabel('\boldmath$f_1(\epsilon_2,\epsilon_3)$','Interpreter','latex');
title(['min f_1 st f_2\leq\epsilon_2, f_3\leq\epsilon_3;     mpASM split '
    num2str(split)]);
```

```
103 %(f2, f3, f1)
104 plot3(0.194,0.605,0.012, 's','MarkerSize',14,'MarkerEdgeColor','white',...
105     'MarkerFaceColor','black');
106 plot3(0.194,0.605,0.012, 's','MarkerSize',10,'MarkerEdgeColor','black',...
107     'MarkerFaceColor',grey);
108 text(0.1953,0.5054,0.01164,'A');
109 plot3(0.016, 0.203,0.207, 's','MarkerSize',14,'MarkerEdgeColor','white',...
110     'MarkerFaceColor','black');
111 plot3(0.016, 0.203,0.207, 's','MarkerSize',10,'MarkerEdgeColor','black',...
112     'MarkerFaceColor',grey);
113 text(0.01, 0.2,0.2,'B','color','black');
114 plot3( 0.038, 0.195, 0.116 , 's','MarkerSize',14,'MarkerEdgeColor','white',...
115     'MarkerFaceColor','black');
116 plot3( 0.038, 0.195, 0.116 , 's','MarkerSize',10,'MarkerEdgeColor','black',...
117     'MarkerFaceColor',grey);
118 annotation('textbox',[0.69 0.36 0.045 0.05],'String','C',...
119     'EdgeColor','white','BackgroundColor','white','FaceAlpha',0.8,...
120     'FitBoxToText','on','Margin',0,'HorizontalAlignment','center',...
121     'VerticalAlignment','middle');
122 plot3(0,0,1.933, 's','MarkerSize',14,'MarkerEdgeColor','white',...
123     'MarkerFaceColor','black');
124 plot3(0,0,1.933, 's','MarkerSize',10,'MarkerEdgeColor','black',...
125     'MarkerFaceColor',grey);
126 text(-0.01,0.05,1.933,'D','color','black');
127 % rotating the figure with Pubudu, we decided to use azimuth=-169.5,
128 % elevation=31.6
129 view(-175.9,22.8);
130 hold off;
131 end
132
133
134 %make a grid pattern of the actual solution function, heavier closer to
135 %zero, sparser closer to {eps2Max,eps3Max}
```

```
numPoints=21;
lambda=zeros(2,numPoints);
lambda(1,:)=logspace(-3,log10(eps2Max),numPoints);
lambda(2,:)=logspace(-3,log10(eps3Max),numPoints);

fGrid=zeros(numPoints,numPoints);
x0=mpsaMOP.lb;

options = optimoptions('fmincon','Algorithm','sqp','Display','Off');

for h=1:length(lambda)
    for i=1:length(lambda)
        A=mpsaMOP.A;
        b=mpsaMOP.b+mpsaMOP.E*[lambda(1,h);lambda(2,i)];
        obj=@(x) mpsaMOP.objective(x,[lambda(1,h),lambda(2,i)]);
        const=@(x) mpsaMOP.constraints(x,[lambda(1,h),lambda(2,i)]);

        %hth column, fill the rows
        [temp,fGrid(i,h),~,~,tempDual,~,~]=fmincon(obj,...
            x0,A,b,[],[],mpsaMOP.lb,mpsaMOP.ub,const,options);
    end %for
end

[lambda1, lambda2]=meshgrid(lambda(1,:),lambda(2,:));

figure;    %mpASM
%change the font to Times New Roman
set(0,'DefaultAxesFontName','Times New Roman');
set(0,'DefaultAxesFontSize',12);
set(0,'DefaultTextFontName','Times New Roman');
set(0,'DefaultTextFontSize',12);
set(0,'DefaultAxesFontWeight','normal');
set(0,'DefaultTextFontWeight','normal');
```

```
dotColor='black';
hold on;
surf(lambda1,lambda2,fGrid);
plotParametricFunction(f01ASM,CR01ASM);
xlabel('\boldmath$\epsilon_2$','Interpreter','latex');
ylabel('\boldmath$\epsilon_3$','Interpreter','latex');
zlabel('\boldmath$f_1(\epsilon_2,\epsilon_3)$','Interpreter','latex');
title(['min f_1 st f_2\leq\epsilon_2, f_3\leq\epsilon_3;     mpASM split '
    num2str(split)]);
%(f2, f3, f1)
plot3(0.194,0.605,0.012, 's','MarkerSize',10,'MarkerEdgeColor','white',...
    'MarkerFaceColor','black');
text(0.1953,0.5054,0.01164,'A');
plot3(0.016, 0.203,0.207, 's','MarkerSize',10,'MarkerEdgeColor','white',...
    'MarkerFaceColor','black');
text(0.01, 0.2,0.2,'B','color','black');
plot3( 0.038, 0.195, 0.116 , 's','MarkerSize',10,'MarkerEdgeColor','white',...
    'MarkerFaceColor','black');
text( 0.05, 0.1937, 0.3 ,'C','color','white');
plot3(0,0,1.933, 's','MarkerSize',10,'MarkerEdgeColor','white',...
    'MarkerFaceColor','black');
text(-0.01,0.05,1.933,'D','color','black');
% rotating the figure with Pubudu, we decided to use azimuth=-169.5,
% elevation=31.6
view(-175.9,22.8);
hold off;

%compute points that are lying on the Pareto set; falsed out until
%necessary for ease of running
if false
%check
%f_2(x*(epsilon_2, epsilon_3))= epsilon_2:
%for each CR, try to compute the points \epsilon at which both f are active
```

```
options=optimoptions('fsolve','Display','off');
xParetoRegion=cell(0);
f1ParetoRegion=cell(0);
CRPareto=[];
epsPareto=[];
problemIndices=[];
gurke=1;
for i=1:length(CR01ASM)
    %for each element of lambda, compute f_i(x(lambda))
    systSolve=@(epsilon) [f2(x01ASM{i}*[epsilon,1]')-epsilon(1), f3(...
                          x01ASM{i}*[epsilon,1]')-epsilon(2)];
    for idx=1:1      %3 is the number of vertices in each CR
        eps0=CR01ASM(i).V(idx,:);

        [activeEps,~,exitFlag,~]=fsolve(systSolve,eps0,options);
        res2=abs(f2(x01ASM{i}*[activeEps,1]')-activeEps(1));
        res3=abs(f3(x01ASM{i}*[activeEps,1]')-activeEps(2));

        %only keep the numbers if the constraint norms are close enough
        tol=0.00001;

        if res2<tol && res3<tol && activeEps(1)>0 && activeEps(2)>0
            %the epsilons are a solution; now find where they are located
            for j=1:length(CR01ASM)
                if CR01ASM(j).contains(activeEps')
                    xParetoRegion{gurke,1}=x01ASM{j};
                    f1ParetoRegion{gurke,1}=f01ASM{j};
                    CRPareto=[CRPareto; CR01ASM(j)];
                    epsPareto=[epsPareto; activeEps];
                    gurke=gurke+1;
                    break; %a CR containing this pareto point was found;
                           %stop looking
                end
```

```
234                 end
235             end
236         end %vertex for
237 end
238
239 gurke=gurke-1;
240 %note that there are x1,x2,x3 and x4, x5, x6 (the absolute value vars)
241 xPareto=zeros(gurke,6);
242 f1Pareto=zeros(gurke,1);
243 for i=1:gurke
244     xPareto(i,:)=xParetoRegion{i}*[epsPareto(i,:),1]';
245     f1Pareto(i)=f1ParetoRegion{i}*[epsPareto(i,:),1]';
246 end
247 figure;
248 plotParametricFunction(f1ParetoRegion,CRPareto);
249 plot3(epsPareto(:,1),epsPareto(:,2),f1Pareto,'o','MarkerSize',10,...
250     'MarkerFaceColor','g');
251 xlabel('\epsilon_2');
252 ylabel('\epsilon_3');
253 zlabel('f_1');
254 title(['min f_1 st f_2\leq\epsilon_2, f_3\leq\epsilon_3;' ...
255     newline 'mpASM,    Pareto Dots   \epsilon_2\in[0,' num2str(eps2Max)...
256     '], \epsilon_3\in[0,' num2str(eps3Max) ']']);
257 % %Pubudu's points
258 %  f2       f3        f1
259 %0.1953   0.6054    0.01164
260 %0.03622  0.2639    0.08381
261 % 0.0375   0.1937   0.1109
262 %0        0         1.933
263 plot3(0.1953,0.6054,0.01164, '.k','MarkerSize',75);
264 plot3(0.03622, 0.2639,0.08381, '.k','MarkerSize',75);
265 plot3( 0.0375, 0.1937, 0.1109 , '.k','MarkerSize',75);
266 plot3(0,0,1.933, '.k','MarkerSize',75);
```

```
267
268 %  X(f1)    Y(f2)    Z(f3)
269 %0.01164   0.1953   0.6054
270 %0.08381   0.03622  0.2639
271 %0.1109    0.0375   0.1937
272 %1.933     0        0
273
274 figure;
275 plotParametricFunction(xParetoRegion,CRPareto,[],[],1);
276 plot3(epsPareto(:,1),epsPareto(:,2),xPareto(:,1),'o','MarkerSize',10,...
277     'MarkerFaceColor','g');
278 xlabel('\epsilon_2');
279 ylabel('\epsilon_3');
280 zlabel('x_1');
281 title(['min f_1 st f_2\leq\epsilon_2, f_3\leq\epsilon_3;' ...
282     newline 'mpASM,    Pareto Dots   \epsilon_2\in[0,' num2str(eps2Max)...
283     '], \epsilon_3\in[0,' num2str(eps3Max) ']']);
284
285 figure;
286 plotParametricFunction(xParetoRegion,CRPareto,[],[],2);
287 plot3(epsPareto(:,1),epsPareto(:,2),xPareto(:,2),'o','MarkerSize',10,...
288     'MarkerFaceColor','g');
289 xlabel('\epsilon_2');
290 ylabel('\epsilon_3');
291 zlabel('x_2');
292 title(['min f_1 st f_2\leq\epsilon_2, f_3\leq\epsilon_3;' ...
293     newline 'mpASM,    Pareto Dots   \epsilon_2\in[0,' num2str(eps2Max)...
294     '], \epsilon_3\in[0,' num2str(eps3Max) ']']);
295
296 figure;
297 plotParametricFunction(xParetoRegion,CRPareto,[],[],3);
298 plot3(epsPareto(:,1),epsPareto(:,2),xPareto(:,3),'o','MarkerSize',10,...
299     'MarkerFaceColor','g');
```

```
300 xlabel('\epsilon_2');
301 ylabel('\epsilon_3');
302 zlabel('x_3');
303 title(['min f_1 st f_2\leq\epsilon_2, f_3\leq\epsilon_3;' ...
304     newline 'mpASM,    Pareto Dots   \epsilon_2\in[0,' num2str(eps2Max)...
305     '], \epsilon_3\in[0,' num2str(eps3Max) ']']);
306
307 end
```

```
1 %the nonlinear constraint for nonSquareTOP
2 function [g, h] = nonlcon(x,theta,func)
3     g(1)=func(x)-theta(1);
4     h=[];
5 end
```

B.4 Portfolio Example

The code for Section 2.5.2 is given below.

```
1 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      Matlab codes\mp ASM algorithm\mpASM')
2 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      MPT Practice\mpSAPractice')
3
4
5 Q1= [1, 0, -1; 0, 2, 0; -1, 0, 2.5];
6 Q2= [3, -1, 0; -1, 4, 1; 0, 1, 3.5];
7
8 %12-18-2020: p3=[-13.5;20; theta];
9 % p3=[-13.5; 20; 16];
10 p3=[-13.5; 20; 0];
11
12 %x1+x2+x3=1
13 % A=[1,1,1;-1,-1,-1];
```

```
14 % b=[1;-1];
15 % Aeq=[1,1,1];
16 % beq=1;
17
18 lb= zeros(3,1);
19 ub= inf*ones(3,1);
20
21 %initial guess
22 x0=lb;
23
24 %must receive column x;
25 f1=@(x) 0.5*x'*Q1*x;
26 f2=@(x) 0.5*x'*Q2*x;
27 f3=@(x) -p3'*x;
28
29 %no nonlinear constraints
30 const=[];
31
32 options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
33
34 %optimization problem structure
35 mpsaMOP = struct;
36
37 %nonlinear constraints (for error computation, only)
38 mpsaMOP.constraints=[];
39
40 %constraints to be relaxed; there are none
41 % mpsaMOP.relConIneq{1}=@(x,gamma) f1(x)-gamma(1);
42 % mpsaMOP.relConEq=cell(0);
43
44 %lower and upper bounds for x
45 mpsaMOP.lb = lb;
46 mpsaMOP.ub = ub;
```

```
47
48 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
49 %weighted sum is weird for mpASM; let's try epsilon-constraints; this has
50 %the added bonus of being able to use mp-SA; we'll make epsilon_1 be for
51 %the function f_1 and epsilon_2 for the function f_2 and then minimize with
52 %respect to f_3
53 Aeq=[1,1,1];
54 beq=1;
55 [x1Hat,f1Hat]=fmincon(f1,lb,[],[],Aeq,beq,lb,ub,const,options);
56 [x2Hat,f2Hat]=fmincon(f2,lb,[],[],Aeq,beq,lb,ub,const,options);
57 [x3Hat,f3Hat]=fmincon(f3,lb,[],[],Aeq,beq,lb,ub,const,options);
58
59 epsilon1min=f1Hat;
60 epsilon1max=max(f1(x2Hat),f1(x3Hat));
61 % epsilon1max=min(f1(x2Hat),f1(x3Hat));
62
63 epsilon2min=f2Hat;
64 epsilon2max=max(f2(x1Hat),f2(x3Hat));
65 % epsilon2max=min(f2(x1Hat),f2(x3Hat));
66
67 %10-31-2020: normalize epsilon1 and epsilon2
68 f1norm=@(x) (0.5*x'*Q1*x-epsilon1min)/(epsilon1max-epsilon1min);
69 f2norm=@(x) (0.5*x'*Q2*x-epsilon2min)/(epsilon2max-epsilon2min);
70
71 % thetaMin=15;
72 % thetaMax=17;
73 thetaMin=0;
74 thetaMax=1;
75 Lambda = Polyhedron('lb',[epsilon1min,epsilon2min,thetaMin],'ub',...
76     [epsilon1max,epsilon2max,thetaMax]);
77 % Lambda = Polyhedron('lb',[0,0],'ub',[1,1]);
78
79 % 12-21-2020: normalize theta:=lambda(3) 2*((theta-15)/2+15/2)
```

```
80  mpsaMOP.objective = @(x,lambda) f3(x)-(2*lambda(3)+15)*x(3);
81
82  % mpsaMOP.objective = @(x,lambda) f3(x)-lambda(3)*x(3);
83
84  %nonlinear constraints (for error computation, only)
85  mpsaMOP.constraints=@(x,epsilon) nonlconEps(x,epsilon,f1,f2);
86  % mpsaMOP.constraints=@(x,epsilon) nonlconEps(x,epsilon,f1norm,f2norm);
87  % mpsaMOP.constraints=[];
88
89  %linear constraint matrices (inequality and equality)
90  mpsaMOP.A = [];
91  mpsaMOP.E = [];
92  mpsaMOP.b = [];
93  mpsaMOP.Aeq = [1,1,1];
94  mpsaMOP.beq = [1];
95  mpsaMOP.Eeq = [0,0,0];
96
97  cap=5;
98  [x01ASMeps,f01ASMeps,CR01ASMeps,solnInfo01ASMeps] =...
99      mpASM_preLEM(0.01,Lambda,mpsaMOP,cap);
100
101 %test polyhedra for 'PortfolioParamReturnEps'
102 % gurkePoly=Polyhedron([0.5448,1.256,0.5;0.1265,1.944,0.5;0.1265,0.5766,0.5]);
103 % gurkePoly=Polyhedron([0.15,0.7,0.5;0.5,1.257,0.5;0.15,1.85,0.5]);
104 % gurkeBool=CR01ASMeps.contains(gurkePoly);
105
106 %regions 1 and 5 are the top butterfly
107 %regions 46 and 50 are the bottom butterfly
108 %note that regions [1:5] and [45:50] share exact overlap for two regions
109 %and are close for a third  (46 and 47: one CR is a subset of the other)
110 %(49 and 50: one CR is a subset of the other)
111
112 %Note that CR01ASMeps(i) do not actually contain \theta=0.5 over their
```

```
%entire domains, so the calculation is apparently being skewed. I think
%that this is a plotParametricFunction issue, so I will have to see
%1-19-21: it was; changing a projection to a slice worked (for now)
% figure
% for i=[1,5,46,50]
%% for i=40:length(CR01ASMeps)
%% for i=[1,5]
%%      gurkeBool(i)=CR01ASMeps(i).contains(gurkePoly);
%%      figure
%      hold on;
%      plot(CR01ASMeps(i));
%%      plot(gurkePoly);
%%      plotParametricFunction(f01ASMeps(i),CR01ASMeps(i),[1,2],0.5);
%      hold off;
% end
% xlabel('\epsilon_1')
% ylabel('\epsilon_2')
% figure
% for i=[1,5,46,50]
%%      gurkeBool(i)=CR01ASMeps(i).contains(gurkePoly);
%%      figure
%      hold on;
%%      plot(CR01ASMeps(i));
%      plotParametricFunction(f01ASMeps(i),CR01ASMeps(i),[1,2],0.5);
%      hold off;
% end
% xlabel('\epsilon_1')
% ylabel('\epsilon_2')

%change the font to Times New Roman
set(0,'DefaultAxesFontName','Times New Roman');
set(0,'DefaultAxesFontSize',12);
set(0,'DefaultTextFontName','Times New Roman');
```

```
146 set(0,'DefaultTextFontSize',12);
147
148 fig1=figure;
149 hold on      %works better with SQP for fmincon
150 %plot against eps1 and eps2 while setting theta=15
151 % plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],15);
152 plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],0);
153 title('Portfolio \epsilon-constraint mp-ASM, \theta=15 normalized wrt \theta')
154 % run('Obj_space_Weight_sum.m')
155 xlabel('\epsilon_1');
156 ylabel('\epsilon_2');
157 zlabel('f_3(\epsilon_1,\epsilon_2,15)');
158 % theta=15
159 % D: [0.12,0.6496,-0.36]
160 plot3(0.12,0.6496,-0.36,'s','MarkerSize',10,'MarkerEdgeColor','white',...
161     'MarkerFaceColor','black');
162 text(0.12,0.8,0,'D');
163 % C: [0.1728,0.5299,-3.455]
164 plot3(0.1728,0.5299,-3.455,'s','MarkerSize',10,'MarkerEdgeColor','white',...
165     'MarkerFaceColor','black');
166 text(0.19,0.5299,-1.75,'C');
167 % B: [0.5588,1.193,-17.59]
168 plot3(0.5588,1.193,-17.59,'s','MarkerSize',10,'MarkerEdgeColor','white',...
169     'MarkerFaceColor','black');
170 text(0.55,1.1,-15.8,'B');
171 % A: [1,2,-20]
172 plot3(1,2,-20,'s','MarkerSize',10,'MarkerEdgeColor','white',...
173     'MarkerFaceColor','black');
174 text(0.99,1.9,-18,'A');
175 %default azimuth is -37.5; to rotate, use 180-37.5=142.5; default elevation
176 %is 30; keep that
177 view([142.5,30]);
178 % zlim([-20,0]);
```

```
179 hold off
180 %change the width of the figure to be the height of the figure (make
181 %shorter butterfly wings)
182 fig1.Position(3)=fig1.Position(4);
183 wahx=gca;
184 set(wahx,'linewidth',1);
185
186 fig2=figure;
187 hold on      %works better with SQP for fmincon
188 %plot against eps1 and eps2 while setting theta=16
189 % plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],16);
190 plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],0.5);
191 title('Portfolio \epsilon-constraint mp-ASM, \theta=16')
192 % run('Obj_space_Weight_sum.m')
193 xlabel('\epsilon_1');
194 ylabel('\epsilon_2');
195 zlabel('f_3(\epsilon_1,\epsilon_2,16)');
196 % zlim([-20,0]);
197 % theta=16
198 % D: [0.12,0.6496,0.-0.02813]
199 plot3(0.12,0.6496,0.-0.02813,'s','MarkerSize',10,'MarkerEdgeColor','white',...
200     'MarkerFaceColor','black');
201 text(0.12,0.84,0,'D');
202 % C: [0.1728,0.5299,-3.664]
203 plot3(0.1728,0.5299,-3.664,'s','MarkerSize',10,'MarkerEdgeColor','white',...
204     'MarkerFaceColor','black');
205 text(0.19,0.5299,-1.9,'C');
206 % B: [0.5532,1.185,-17.99]
207 plot3(0.5532,1.185,-17.99,'s','MarkerSize',10,'MarkerEdgeColor','white',...
208     'MarkerFaceColor','black');
209 text(0.5,1.2,-15.9,'B');
210 % A: [1,2,-20]
211 plot3(1,2,-20,'s','MarkerSize',10,'MarkerEdgeColor','white',...
```

```
212     'MarkerFaceColor','black');
213 text(0.99,1.9,-19,'A');
214 %default azimuth is -37.5; to rotate, use 180-37.5=142.5; default elevation
215 %is 30; keep that
216 view([142.5,30]);
217 hold off
218 %change the width of the figure to be the height of the figure (make
219 %shorter butterfly wings)
220 fig2.Position(3)=fig2.Position(4);
221
222 fig3=figure;
223 hold on      %works better with SQP for fmincon
224 %plot against eps1 and eps2 while setting theta=17
225 % plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],17);
226 plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],1);
227 title('Portfolio \epsilon-constraint mp-ASM, \theta=17')
228 % run('Obj_space_Weight_sum.m')
229 xlabel('\epsilon_1');
230 ylabel('\epsilon_2');
231 zlabel('f_3(\epsilon_1,\epsilon_2,17)');
232 % zlim([-20,0]);
233 % theta=17
234 % D: [0.12,0.6464,-0.3475]
235 plot3(0.12,0.6464,-0.3475,'s','MarkerSize',10,'MarkerEdgeColor','white',...
236     'MarkerFaceColor','black');
237 text(0.12,0.84,0,'D');
238 % C: [0.1715,0.5299,-3.869]
239 plot3(0.1715,0.5299,-3.869,'s','MarkerSize',10,'MarkerEdgeColor','white',...
240     'MarkerFaceColor','black');
241 text(0.19,0.5299,-2.1,'C');
242 % B: [0.5607,1.187,-18.49]
243 plot3(0.5607,1.187,-18.49,'s','MarkerSize',10,'MarkerEdgeColor','white',...
244     'MarkerFaceColor','black');
```

```
text(0.55,1.1,-17,'B');
% A: [1,2,-20]
plot3(1,2,-20,'s','MarkerSize',10,'MarkerEdgeColor','white',...
    'MarkerFaceColor','black');
text(0.99,1.9,-18,'A');
%default azimuth is -37.5; to rotate, use 180-37.5=142.5; default elevation
%is 30; keep that
view([142.5,30]);
hold off
%change the width of the figure to be the height of the figure (make
%shorter butterfly wings)
fig3.Position(3)=fig3.Position(4);

%extracting data points (1-30-2021)
% \epsilon_1=0.517, \epsilon_2=1.445, \theta=16
wahX=x01ASMeps{CR01ASMeps.contains(Polyhedron([0.517, 1.445, 0.5]))};
f01ASMeps{CR01ASMeps.contains(Polyhedron([0.517, 1.445, 0.5]))};
wahPoly=CR01ASMeps(CR01ASMeps.contains(Polyhedron([0.517, 1.445,...
    0.5])));
wahPoly.H

% %x1+x2+x3=1
% mpsaMOP.A = [];
% mpsaMOP.b = [];
% mpsaMOP.E = [];
% mpsaMOP.Aeq = [1,1,1];
% mpsaMOP.Eeq = [0,0];
% mpsaMOP.beq = [1];
%
% %constraints to be relaxed;          %10-31-2020: normalize
% % mpsaMOP.relConIneq{1}=@(x,epsilon) f1(x)-epsilon(1);
% % mpsaMOP.relConIneq{2}=@(x,epsilon) f2(x)-epsilon(2);
% mpsaMOP.relConIneq{1}=@(x,epsilon) f1norm(x)-epsilon(1);
```

```
278 % mpsaMOP.relConIneq{2}=@(x,epsilon) f2norm(x)-epsilon(2);
279 %
280 % mpsaMOP.relConEq=cell(0);
281 %
282 % %unrelaxed constraints
283 % mpsaMOP.baseConstraints=[];
284 %
285 % %lower and upper bounds for x
286 % mpsaMOP.lb = lb;
287 % mpsaMOP.ub = ub;
288 %
289 % %initial guess for dual variables: v does not exist
290 % %dual variables are one cell per CR, with matrix rows representing
291 % %affine combinations of theta
292 %
293 % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
294 % %let's try to write a program to compute the initial u0, since guessing
295 % %seems to be mucking things up
296 % %the polyhedron is not a simplex, so triangulate it, take one simplex,
297 % %and run with it.
298 % tempSimplex=Lambda.triangulate.V;
299 % vertices=tempSimplex;
300 %
301 % uStart=[];
302 % %at each vertex, compute the solution and get the dual variable;
303 % x0=mpsaMOP.lb;
304 % % A=mpsaMOP.A;
305 % Aeq=mpsaMOP.Aeq;
306 % beq=mpsaMOP.beq;
307 % options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
308 % for i=1:length(vertices)
309 % %       tempB=mpsaMOP.b+mpsaMOP.E*vertices(i,:)';
310 %       tempBeq=mpsaMOP.beq+mpsaMOP.Eeq*vertices(i,:)';
```

```
%        obj=@(x) mpsaMOP.objective(x,vertices(i,:));
%        const=@(x) mpsaMOP.constraints(x,vertices(i,:));
%%         [~,~,~,~,tempU,~,~]=fmincon(obj,x0,A,tempB,[],[],...
%%              mpsaMOP.lb,mpsaMOP.ub,const,options);
%        [~,~,~,~,tempU,~,~]=fmincon(obj,x0,[],[],Aeq, tempBeq,...
%             mpsaMOP.lb,mpsaMOP.ub,const,options);
%        uStart=[uStart;tempU.ineqnonlin'];
% end %for
%
% uNorm=norm(uStart,2);
% if uNorm>1
%        uStart=uStart./uNorm;
% end
%
% %compute the linear interpolation for u0
% M=[vertices, ones(length(vertices),1)];
% Minv=M^-1;
% %transpose so that we have each row being the interpolation
% uInterp= (Minv*uStart)';
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% u0{1}=uInterp;
%
% v0={};
%
% [x01,u01,v01,f01,CR01,solnInfo01]=mpSA(u0,v0,0.1,Lambda,mpsaMOP,10,cap);
% figure
% hold on
% title('Portfolio \epsilon-constraint mp-SA')
% % run('Obj_space_Weight_sum.m')
% xlabel('\epsilon_1');
% ylabel('\epsilon_2');
% hold off
% plotParametricFunction(f01,CR01);
```

```
344
345 %5-5-2020: this means that shallowness of the objective function is
346 %correlated to the ability of ASM
347 % [x1Hat,f1Hat]=fmincon(f1,lb,A,b,[],[],lb,ub,const,options);
348 % [x2Hat,f2Hat]=fmincon(f2,lb,A,b,[],[],lb,ub,const,options);
349 %
350 % f1Cross=f1(x2Hat);  %f1 at the minimizer of f2
351 % f2Cross=f2(x1Hat);  %f2 at the minimizer of f1
352 %
353 % %plot the endpoints
354 % hold on
355 % plot([f1Hat],[f2Cross],'bo')
356 % plot([f1Cross],[f2Hat],'ko')
357 % xlabel('f_1');
358 % ylabel('f_2');
359 % xtickformat('%.2f')
360 % legend('(f1Hat, f2Cross)', '(f1Cross, f2Hat)')
361 % hold off
362 %
363 % %take different values of f1
364 % gamma=linspace(f1Hat,f1Cross,8);
365 % x2grid=cell(0);
366 % f2grid=[];
367 % for i=1:length(gamma)
368 %     f1const=@(x) nonlcon(x,gamma(i),f1);
369 %     [tempSoln,tempVal]=fmincon(f2,x0,A,b,[],[],lb,ub,f1const);
370 %     x2grid{i}=tempSoln;
371 %     f2grid(i)=tempVal;
372 % end %for
373 % plot(gamma,f2grid,'bx-');
374 % xlabel('\gamma');
375 % ylabel('f_2^*');
376 % title('min f_2 st f_1\leq\gamma, Ax\leq b');
```

```
%the nonlinear constraint for epsilon-constraint triObjPortfolio
function [g, h] = nonlconEps(x,epsilon,f1,f2)
    g(1)=f1(x)-epsilon(1);
    g(2)=f2(x)-epsilon(2);
    h=[];
end
```

```
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm\mpASM')
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice')

%make sure we have -theta

%figure out how to plot

Q1= [1, 0, -1; 0, 2, 0; -1, 0, 2.5];
Q2= [3, -1, 0; -1, 4, 1; 0, 1, 3.5];

%12-18-2020: p3=[-13.5;20; theta];
% p3=[-13.5; 20; 16];
p3=[-13.5; 20; 0];

%x1+x2+x3=1
% A=[1,1,1;-1,-1,-1];
% b=[1;-1];
% Aeq=[1,1,1];
% beq=1;

lb= zeros(3,1);
ub= inf*ones(3,1);

%initial guess
```

```
25 x0=lb;
26
27 %must receive column x;
28 f1=@(x) 0.5*x'*Q1*x;
29 f2=@(x) 0.5*x'*Q2*x;
30 f3=@(x) -p3'*x;     %minimize the negative return
31 % f3=@(x) p3'*x;     %maximize the return
32
33 %no nonlinear constraints
34 const=[];
35
36 options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
37
38 %optimization problem structure
39 mpsaMOP = struct;
40
41 %nonlinear constraints (for error computation, only)
42 mpsaMOP.constraints=[];
43
44 %constraints to be relaxed; there are none
45 % mpsaMOP.relConIneq{1}=@(x,gamma) f1(x)-gamma(1);
46 % mpsaMOP.relConEq=cell(0);
47
48 %lower and upper bounds for x
49 mpsaMOP.lb = lb;
50 mpsaMOP.ub = ub;
51
52 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
53 Aeq=[1,1,1];
54 beq=1;
55
56 f1norm=@(x) (0.5*x'*Q1*x-epsilon1min)/(epsilon1max-epsilon1min);
57 f2norm=@(x) (0.5*x'*Q2*x-epsilon2min)/(epsilon2max-epsilon2min);
```

```
58
59 % thetaMin=15;
60 % thetaMax=17;
61 thetaMin=0;
62 thetaMax=1;
63
64 % epsilonMin=-20;
65 % epsilonMax=13.5;
66 epsilonMin=0;
67 epsilonMax=1;
68 %lambda(1):= weighted sum parameter
69 %lambda(2):= epsilon-constraint parameter
70 %lambda(3):= return parameter
71 Lambda = Polyhedron('lb',[0,epsilonMin,thetaMin],'ub',...
72     [1,epsilonMax,thetaMax]);
73
74 %1-26-21: minimize the weighted sum of the risks such that the negative
75 %return doesn't go above a parameter
76 mpsaMOP.objective = @(x,lambda) lambda(1)*f1(x) +(1-lambda(1))*f2(x);
77
78 mpsaMOP.constraints=@(x,epsilon) nonlconModHyb(x,epsilon);
79 % mpsaMOP.constraints=[];
80
81 %linear constraint matrices (inequality and equality)
82 mpsaMOP.A = [];
83 mpsaMOP.E = [];
84 mpsaMOP.b = [];
85 mpsaMOP.Aeq = [1,1,1];
86 mpsaMOP.beq = [1];
87 mpsaMOP.Eeq = [0,0,0];
88
89 % tic
90 cap=5;
```

```
91  [x01ASMeps,f01ASMeps,CR01ASMeps,solnInfo01ASMeps] =...
92      mpASM_preLEM(0.01,Lambda,mpsaMOP,cap);
93
94  figure
95  hold on      %works better with SQP for fmincon
96  %plot against eps1 and eps2 while setting theta=15
97  % plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],15);
98  plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],0);
99  title(['Portfolio modified-hybrid-constraint mp-ASM, \theta=15 normalized wrt
        \theta'...
100     ' splitDepth=' num2str(cap)])
101 % run('Obj_space_Weight_sum.m')
102 xlabel('\lambda');
103 ylabel('\epsilon');
104 zlabel('MWR(\lambda,\epsilon,15)');
105 % zlim([-20,0]);
106 hold off
107 % time=toc;
108
109 figure
110 hold on      %works better with SQP for fmincon
111 %plot against eps1 and eps2 while setting theta=16
112 % plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],16);
113 plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],0.5);
114 title(['Portfolio modified-hybrid-constraint mp-ASM, \theta=16'...
115     ' splitDepth=' num2str(cap)])
116 % run('Obj_space_Weight_sum.m')
117 xlabel('\lambda');
118 ylabel('\epsilon');
119 zlabel('MWR(\lambda,\epsilon,16)');
120 % zlim([-20,0]);
121 hold off
122
```

```
figure
hold on      %works better with SQP for fmincon
%plot against eps1 and eps2 while setting theta=16
% plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],16);
plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],1);
title(['Portfolio modified-hybrid-constraint mp-ASM, \theta=17'...
    ' splitDepth=' num2str(cap)])
% run('Obj_space_Weight_sum.m')
xlabel('\lambda');
ylabel('\epsilon');
zlabel('MWR(\lambda,\epsilon,17)');
% zlim([-20,0]);
hold off

%checking if theta is unnecessary
figure
hold on      %works better with SQP for fmincon
plotParametricFunction(f01ASMeps,CR01ASMeps,[2,3],0);
title(['Portfolio modified-hybrid-constraint mp-ASM, \lambda=0'...
    ' splitDepth=' num2str(cap)])
% run('Obj_space_Weight_sum.m')
xlabel('\epsilon');
ylabel('\theta');
zlabel('MWR(0,\epsilon,\theta)');
% zlim([-20,0]);
hold off
% time=toc;

figure
hold on      %works better with SQP for fmincon
plotParametricFunction(f01ASMeps,CR01ASMeps,[2,3],0.25);
title(['Portfolio modified-hybrid-constraint mp-ASM, \lambda=0.25'...
    ' splitDepth=' num2str(cap)])
```

```
% run('Obj_space_Weight_sum.m')
xlabel('\epsilon');
ylabel('\theta');
zlabel('MWR(0.25,\epsilon,\theta)');
% zlim([-20,0]);
view(134.5,31.6);
hold off

figure
hold on      %works better with SQP for fmincon
plotParametricFunction(f01ASMeps,CR01ASMeps,[2,3],0.5);
title(['Portfolio modified-hybrid-constraint mp-ASM, \lambda=0.5'...
    ' splitDepth=' num2str(cap)])
% run('Obj_space_Weight_sum.m')
xlabel('\epsilon');
ylabel('\theta');
zlabel('MWR(0.5,\epsilon,\theta)');
% zlim([-20,0]);
view(134.5,31.6);
hold off

figure
hold on      %works better with SQP for fmincon
plotParametricFunction(f01ASMeps,CR01ASMeps,[2,3],0.75);
title(['Portfolio modified-hybrid-constraint mp-ASM, \lambda=0.75'...
    ' splitDepth=' num2str(cap)])
% run('Obj_space_Weight_sum.m')
xlabel('\epsilon');
ylabel('\theta');
zlabel('MWR(0.75,\epsilon,\theta)');
% zlim([-20,0]);
view(134.5,31.6);
hold off
```

```
189
190 figure
191 hold on      %works better with SQP for fmincon
192 plotParametricFunction(f01ASMeps,CR01ASMeps,[2,3],1);
193 title(['Portfolio modified-hybrid-constraint mp-ASM, \lambda=1'...
194     ' splitDepth=' num2str(cap)])
195 % run('Obj_space_Weight_sum.m')
196 xlabel('\epsilon');
197 ylabel('\theta');
198 zlabel('MWR(1,\epsilon,\theta)');
199 % zlim([-20,0]);
200 hold off
201
202
203 %we got an answer, but we need to evaluate/graph f1(x01ASM),f2(x01ASM), and
204 %also f3(x01ASM) at all the parameters to see how accurate mpASM is
205 f1plot=[];
206 f2plot=[];
207 f3plot=[];
208
209 %consider theta=16, since that's the one we have
210 thetaVal=0.5;
211 CRASM=slice(CR01ASMeps,3,0.5,'keepDim',false);
212 %for each CR, apply the grid function, and then evaluate f_i(x(lambda))
213 for i=1:length(CRASM)
214     %for each element of lambda, compute f_i(x(lambda))
215     lambda=CRASM(i).V;
216 %       [lambda1,lambda2]=meshgrid(lambda);
217     for j=1:length(lambda)
218 %       for k=1:length(lambda2)
219         xIJ=x01ASMeps{i}*[lambda(j,1:2),thetaVal,1]';
220         f1plot=[f1plot,f1(xIJ)];
221         f2plot=[f2plot,f2(xIJ)];
```

```
222             f3Temp=13.5*xIJ(1)-20*xIJ(2)-(2*thetaVal+15)*xIJ(3);
223             f3plot=[f3plot,f3Temp];
224 %                   surf(f1plot,f2plot,f3plot);
225 %       end %for k
226       end %for j
227 end %for i
228 open('ObjectiveSpace_theta_16_on_2_12.fig')
229 hold on
230 plot3(f1plot,f2plot,f3plot,'s','MarkerSize',12,'MarkerEdgeColor','black')%,...
231 %       'MarkerFaceColor','white');
232 % run('Obj_space_Weight_sum.m')
233 xlabel('$f_1({\bf x};16)$','Interpreter','latex');
234 ylabel('$f_2({\bf x};16)$','Interpreter','latex');
235 zlabel('$f_3({\bf x};16)$','Interpreter','latex');
236 xlim([0,1]);
237 zlim([-20,0]);
238 title(['Pareto Points modified hybrid Portfolio Example using mpASM']);
239 % view(126.5,14);
240 hold off
```

```
1 %the nonlinear constraint for modified-hybrid-constraint triObjPortfolio
2 function [g, h] = nonlconModHyb(x,epsilon)
3     %this is -p(\theta)^Tx-\epsilon<=0 with \theta and \epsilon normalized
4     g(1)=13.5*x(1)-20*x(2)-(2*epsilon(3)+15)*x(3)-33.5*epsilon(2)+20;
5     h=[];
6 end
```

```
1 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      Matlab codes\mp ASM algorithm\mpASM')
2 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      MPT Practice\mpSAPractice')
3
4
5 Q1= [1, 0, -1; 0, 2, 0; -1, 0, 2.5];
```

```
6  Q2= [3, -1, 0; -1, 4, 1; 0, 1, 3.5];
7
8  %12-18-2020: p3=[-13.5;20; theta];
9  % p3=[-13.5; 20; 16];
10 p3=[-13.5; 20; 0];
11
12 %x1+x2+x3=1
13 % A=[1,1,1;-1,-1,-1];
14 % b=[1;-1];
15 % Aeq=[1,1,1];
16 % beq=1;
17
18 lb= zeros(3,1);
19 ub= inf*ones(3,1);
20
21 %initial guess
22 x0=lb;
23
24 %must receive column x;
25 f1=@(x) 0.5*x'*Q1*x;
26 f2=@(x) 0.5*x'*Q2*x;
27 f3=@(x) -p3'*x;
28
29 %no nonlinear constraints
30 const=[];
31
32 options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
33
34 %optimization problem structure
35 mpsaMOP = struct;
36
37 %nonlinear constraints (for error computation, only)
38 mpsaMOP.constraints=[];
```

```
39
40 %constraints to be relaxed; there are none
41 % mpsaMOP.relConIneq{1}=@(x,gamma) f1(x)-gamma(1);
42 % mpsaMOP.relConEq=cell(0);
43
44 %lower and upper bounds for x
45 mpsaMOP.lb = lb;
46 mpsaMOP.ub = ub;
47
48 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
49 %weighted sum is weird for mpASM; let's try epsilon-constraints; this has
50 %the added bonus of being able to use mp-SA; we'll make epsilon_1 be for
51 %the function f_1 and epsilon_2 for the function f_2 and then minimize with
52 %respect to f_3
53 Aeq=[1,1,1];
54 beq=1;
55 [x1Hat,f1Hat]=fmincon(f1,lb,[],[],Aeq,beq,lb,ub,const,options);
56 [x2Hat,f2Hat]=fmincon(f2,lb,[],[],Aeq,beq,lb,ub,const,options);
57 [x3Hat,f3Hat]=fmincon(f3,lb,[],[],Aeq,beq,lb,ub,const,options);
58
59 epsilon1min=f1Hat;
60 epsilon1max=max(f1(x2Hat),f1(x3Hat));
61 % epsilon1max=min(f1(x2Hat),f1(x3Hat));
62
63 epsilon2min=f2Hat;
64 epsilon2max=max(f2(x1Hat),f2(x3Hat));
65 % epsilon2max=min(f2(x1Hat),f2(x3Hat));
66
67 %10-31-2020: normalize epsilon1 and epsilon2
68 f1norm=@(x) (0.5*x'*Q1*x-epsilon1min)/(epsilon1max-epsilon1min);
69 f2norm=@(x) (0.5*x'*Q2*x-epsilon2min)/(epsilon2max-epsilon2min);
70
71 % thetaMin=15;
```

```
72  % thetaMax=17;
73  thetaMin=0;
74  thetaMax=1;
75  %note that epsilon is limiting the convex combination of f1 and f2 so goes
76  %from min(f2Hat,f3Hat) to max([f1(x2Hat),f1(x3Hat),f2(x1Hat),f2(x3Hat)])
77  epsilonMin=min(f1Hat,f2Hat);
78  epsilonMax=max([f1(x2Hat),f1(x3Hat),f2(x1Hat),f2(x3Hat)]);
79  Lambda = Polyhedron('lb',[0,epsilonMin,thetaMin],'ub',...
80      [1,epsilonMax,thetaMax]);
81
82  % Lambda = Polyhedron('lb',[epsilon1min,epsilon2min,thetaMin],'ub',...
83  %       [epsilon1max,epsilon2max,thetaMax]);
84  % Lambda = Polyhedron('lb',[0,0],'ub',[1,1]);
85
86  % 12-21-2020: normalize theta:=lambda(3) 2*((theta-15)/2+15/2)
87  mpsaMOP.objective = @(x,lambda) f3(x)-(2*lambda(3)+15)*x(3);
88
89  % mpsaMOP.objective = @(x,lambda) f3(x)-lambda(3)*x(3);
90
91  %nonlinear constraints (for error computation, only)
92  mpsaMOP.constraints=@(x,epsilon) nonlconRedEps(x,epsilon,f1,f2);
93  % mpsaMOP.constraints=@(x,epsilon) nonlconEps(x,epsilon,f1norm,f2norm);
94  % mpsaMOP.constraints=[];
95
96  %linear constraint matrices (inequality and equality)
97  mpsaMOP.A = [];
98  mpsaMOP.E = [];
99  mpsaMOP.b = [];
100 mpsaMOP.Aeq = [1,1,1];
101 mpsaMOP.beq = [1];
102 mpsaMOP.Eeq = [0,0,0];
103
104 cap=5;
```

```
105 [x01ASMeps, f01ASMeps, CR01ASMeps, solnInfo01ASMeps] =...
106     mpASM_preLEM(0.01, Lambda, mpsaMOP, cap);
107
108 %change the font to Times New Roman
109 set(0,'DefaultAxesFontName','Times New Roman');
110 set(0,'DefaultAxesFontSize',12);
111 set(0,'DefaultTextFontName','Times New Roman');
112 set(0,'DefaultTextFontSize',12);
113 set(0,'DefaultAxesFontWeight','normal');
114 set(0,'DefaultTextFontWeight','normal');
115
116 figure
117 hold on      %works better with SQP for fmincon
118 %plot against eps1 and eps2 while setting theta=15
119 % plotParametricFunction(f01ASMeps, CR01ASMeps, [1,2], 15);
120 plotParametricFunction(f01ASMeps, CR01ASMeps, [1,2], 0);
121 title(['Portfolio reduced-\epsilon-constraint mp-ASM, \theta=15 normalized wrt
        \theta'...
122     ' splitDepth=' num2str(cap)])
123 % run('Obj_space_Weight_sum.m')
124 xlabel('\boldmath$\lambda$','Interpreter','latex');
125 ylabel('\boldmath$\epsilon$','Interpreter','latex');
126 zlabel('\boldmath$f_3(\lambda,\epsilon,0)$','Interpreter','latex');
127 % zlim([-20,0]);
128 % rotating the figure with Pubudu, we decided to use azimuth=117.7,
129 % elevation=23.6
130 view(117.7,23.6);
131 hold off
132 %thicken the axes
133 wahx=gca;
134 set(wahx,'linewidth',1);
135
136 figure
```

```
137 hold on      %works better with SQP for fmincon
138 %plot against eps1 and eps2 while setting theta=16
139 % plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],16);
140 plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],0.5);
141 title(['Portfolio reduced-\epsilon-constraint mp-ASM, \theta=16'...
142     ' splitDepth=' num2str(cap)])
143 % run('Obj_space_Weight_sum.m')
144 xlabel('\boldmath$\lambda$','Interpreter','latex');
145 ylabel('\boldmath$\epsilon$','Interpreter','latex');
146 zlabel('\boldmath$f_3(\lambda,\epsilon,0.5)$','Interpreter','latex');
147 % zlim([-20,0]);
148 % rotating the figure with Pubudu, we decided to use azimuth=117.7,
149 % elevation=23.6
150 view(117.7,23.6);
151 hold off
152 %thicken the axes
153 wahx=gca;
154 set(wahx,'linewidth',1);
155
156 figure
157 hold on      %works better with SQP for fmincon
158 %plot against eps1 and eps2 while setting theta=17
159 % plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],17);
160 plotParametricFunction(f01ASMeps,CR01ASMeps,[1,2],1);
161 title(['Portfolio reduced-\epsilon-constraint mp-ASM, \theta=17'...
162     ' splitDepth=' num2str(cap)])
163 % run('Obj_space_Weight_sum.m')
164 xlabel('\boldmath$\lambda$','Interpreter','latex');
165 ylabel('\boldmath$\epsilon$','Interpreter','latex');
166 zlabel('\boldmath$f_3(\lambda,\epsilon,1)$','Interpreter','latex');
167 % zlim([-20,0]);
168 % rotating the figure with Pubudu, we decided to use azimuth=117.7,
169 % elevation=23.6
```

```
170 view(117.7,23.6);
171 hold off
172 %thicken the axes
173 wahx=gca;
174 set(wahx,'linewidth',1);
175
176
177 %extracting data points (12-15-2020)
178 % \lambda=0.613, \epsilon=0.6391, \theta=16
179 % wahX=x01ASMeps{CR01ASMeps.contains(Polyhedron([0.613, 0.6391, 0.5]))};
180 % f01ASMeps{CR01ASMeps.contains(Polyhedron([0.613, 0.6391, 0.5]))};
181 % wahPoly=CR01ASMeps(CR01ASMeps.contains(Polyhedron([0.613, 0.6391,...
182 % 0.5])));
183 %wahPoly.H
184
185 % mpsaMOP.objective(x01ASMeps{CR01ASMeps.contains(Polyhedron([0.613,...
186 %      0.6391, 0.5]))}*[0.4169; 0.05169; 0.5; 1],[0.613, 0.6391, 0.5]);
187 %yields -8.5857; makes sense: the program wasn't optimized with this error
188 %in mind
189
190 % %x1+x2+x3=1
191 % mpsaMOP.A = [];
192 % mpsaMOP.b = [];
193 % mpsaMOP.E = [];
194 % mpsaMOP.Aeq = [1,1,1];
195 % mpsaMOP.Eeq = [0,0];
196 % mpsaMOP.beq = [1];
197 %
198 % %constraints to be relaxed;          %10-31-2020: normalize
199 % % mpsaMOP.relConIneq{1}=@(x,epsilon) f1(x)-epsilon(1);
200 % % mpsaMOP.relConIneq{2}=@(x,epsilon) f2(x)-epsilon(2);
201 % mpsaMOP.relConIneq{1}=@(x,epsilon) f1norm(x)-epsilon(1);
202 % mpsaMOP.relConIneq{2}=@(x,epsilon) f2norm(x)-epsilon(2);
```

```
%
% mpsaMOP.relConEq=cell(0);
%
% %unrelaxed constraints
% mpsaMOP.baseConstraints=[];
%
% %lower and upper bounds for x
% mpsaMOP.lb = lb;
% mpsaMOP.ub = ub;
%
% %initial guess for dual variables: v does not exist
% %dual variables are one cell per CR, with matrix rows representing
% %affine combinations of theta
%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %let's try to write a program to compute the initial u0, since guessing
% %seems to be mucking things up
% %the polyhedron is not a simplex, so triangulate it, take one simplex,
% %and run with it.
% tempSimplex=Lambda.triangulate.V;
% vertices=tempSimplex;
%
% uStart=[];
% %at each vertex, compute the solution and get the dual variable;
% x0=mpsaMOP.lb;
% % A=mpsaMOP.A;
% Aeq=mpsaMOP.Aeq;
% beq=mpsaMOP.beq;
% options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
% for i=1:length(vertices)
% %       tempB=mpsaMOP.b+mpsaMOP.E*vertices(i,:)';
%         tempBeq=mpsaMOP.beq+mpsaMOP.Eeq*vertices(i,:)';
%         obj=@(x) mpsaMOP.objective(x,vertices(i,:));
```

```
%       const=@(x) mpsaMOP.constraints(x,vertices(i,:));
% %       [~,~,~,~,tempU,~,~]=fmincon(obj,x0,A,tempB,[],[],...
% %           mpsaMOP.lb,mpsaMOP.ub,const,options);
%       [~,~,~,~,tempU,~,~]=fmincon(obj,x0,[],[],Aeq, tempBeq,...
%           mpsaMOP.lb,mpsaMOP.ub,const,options);
%       uStart=[uStart;tempU.ineqnonlin'];
% end %for
%
% uNorm=norm(uStart,2);
% if uNorm>1
%       uStart=uStart./uNorm;
% end
%
% %compute the linear interpolation for u0
% M=[vertices, ones(length(vertices),1)];
% Minv=M^-1;
% %transpose so that we have each row being the interpolation
% uInterp= (Minv*uStart)';
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% u0{1}=uInterp;
%
% v0={};
%
% [x01,u01,v01,f01,CR01,solnInfo01]=mpSA(u0,v0,0.1,Lambda,mpsaMOP,10,cap);
% figure
% hold on
% title('Portfolio \epsilon-constraint mp-SA')
% % run('Obj_space_Weight_sum.m')
% xlabel('\epsilon_1');
% ylabel('\epsilon_2');
% hold off
% plotParametricFunction(f01,CR01);

```

```
%5-5-2020: this means that shallowness of the objective function is
%correlated to the ability of ASM
% [x1Hat,f1Hat]=fmincon(f1,lb,A,b,[],[],lb,ub,const,options);
% [x2Hat,f2Hat]=fmincon(f2,lb,A,b,[],[],lb,ub,const,options);
%
% f1Cross=f1(x2Hat);  %f1 at the minimizer of f2
% f2Cross=f2(x1Hat);  %f2 at the minimizer of f1
%
% %plot the endpoints
% hold on
% plot([f1Hat],[f2Cross],'bo')
% plot([f1Cross],[f2Hat],'ko')
% xlabel('f_1');
% ylabel('f_2');
% xtickformat('%.2f')
% legend('(f1Hat, f2Cross)', '(f1Cross, f2Hat)')
% hold off
%
% %take different values of f1
% gamma=linspace(f1Hat,f1Cross,8);
% x2grid=cell(0);
% f2grid=[];
% for i=1:length(gamma)
%     f1const=@(x) nonlcon(x,gamma(i),f1);
%     [tempSoln,tempVal]=fmincon(f2,x0,A,b,[],[],lb,ub,f1const);
%     x2grid{i}=tempSoln;
%     f2grid(i)=tempVal;
% end %for
% plot(gamma,f2grid,'bx-');
% xlabel('\gamma');
% ylabel('f_2^*');
% title('min f_2 st f_1\leq\gamma, Ax\leq b');
```

```
%the nonlinear constraint for reduced-epsilon-constraint triObjPortfolio
```

```
function [g, h] = nonlconRedEps(x,epsilon,f1,f2)
    g(1)=epsilon(1)*f1(x)+(1-epsilon(1))*f2(x)-epsilon(2);
    h=[];
end
```

Appendix C Chapter 3 Code

sec:BASMcode

C.1 The Biconvex Approximate Simplex Method

This version of the Approximate Simplex Method contains the two error bound possibilities: the LEM and the BOM; in case the BOM picks a vertex as the error point, it reverts to the LEM.

```
function [xstar,fstar,CR,solnInfo] = mpASM(tol,Omega,problem,varargin)
%This function solves a general mp-NLP using a modification of the simplex
%approximation algorithm of Bemporad and Filippi (2006)
%-------------------------------------------------------------------------
%INPUT              TYPE              DESCRIPTION
%tol:               scalar            error tolerance
%Omega:             polyhedron        parameter space
%problem:           struct            mp-NLP information
%varargin:          cell: scalar      optional cap for CR splits
%                   cell: chars       optional error type: 'BOM'
%                                          'LEM'
%-------------------------------------------------------------------------
%OUTPUT
%xstar:             cell              optimal decision function
%fstar:             cell              optimal value function
%CR:           polyhedron vector      partition of parameter space
%solnInfo:          struct            solution information
%-------------------------------------------------------------------------
%solnInfo Fields:
%error:     sum of largest single point errors for each simplex
%intError:  approximation of integrated error over parameter space
%nlps:      total number of NLP problems solved
```

```
23 %tol:        maximum error tolerance used
24 %numTooSmallPolys:       total number of Polyhedra that were not splittable
25 %time:       number of milliseconds to run
26 %-------------------------------------------------------------------------
27 tic
28 if isempty(varargin)           % Pangia 6-2-2020
29     cap=5;
30     errorType='BOM';
31 elseif length(varargin)==1
32     cap=varargin{1};
33     errorType='BOM';
34 else
35     cap=varargin{1};
36     errorType=varargin{2};
37 end %if
38 %Divide parameter space into simplices
39 S = Omega.triangulate;
40 %=========================================================================
41 %Construct a linear approximate of the solution for each simplex
42 RemainVolume = Omega.volume;
43
44 %1-12-2022: compute the edge lengths of each simplex, and store them as
45 % Data
46 if strcmp(errorType,'LEM')
47 for k=1:length(S)
48     vertices=S(k).V;
49
50     %compute and store the lengths of the vertices
51     numVertices=size(vertices,1);
52     listLen=numVertices*(numVertices-1)/2;
53     lengthsList=zeros(1,listLen);
54     %a list of the row indices of S(k).V corresponding to each edge of the
55     %simplex
```

```
56      vertexOneList=zeros(1,listLen);
57      vertexTwoList=zeros(1,listLen);
58      listPtr=1;
59      for i=1:numVertices-1
60          for j=i+1:numVertices
61              lengthsList(listPtr)=norm(vertices(i,:)-vertices(j,:));
62              vertexOneList(listPtr,:)=i;
63              vertexTwoList(listPtr,:)=j;
64              listPtr=listPtr+1;
65          end
66      end
67      %attach the edge lengths to the simplex
68      S(k).Data={lengthsList,vertexOneList,vertexTwoList};
69  end % edge length for
70  end %if
71
72  [xstar,fstar,CR,solnInfo,~] = mpSimplexApproximation(tol,S,problem,...
73                                  RemainVolume,cap,errorType);
74  solnInfo.tol = tol;
75  solnInfo.time = toc;
76
77
78  %=========================================================
79  end
```

```
1   function [xOpt,fOpt,CRopt,metric,RemainVol] = mpSimplexApproximation(...
2                               tol,S,problem,RemainVol,cap,errorType)
3   %Constructs a linear approximation of the optimal decision and value
4   %functions for each simplex in S. Calculates the maximum error
5   %over each simplex and partitions it further if error is too large.
6   %-------------------------------------------------------------------
7   %INPUT          TYPE                    DESCRIPTION
8   %tol:           scalar                  error tolerance
9   %S:             polyhedron array        set of simplexes
```

```
%problem:          struct                  mp-NLP information
%RemainVol:        scalar                  remaining parameter space to check
%cap:              scalar                  max number of CR splits
%errorType         string                  error method 'BOM' or 'LEM'
%-------------------------------------------------------------------------
%OUTPUT
%xOpt:             cell                    optimal decision function
%fOpt:             cell                    optimal value function
%CRopt:            polyhedron vector       partition of parameter space
%metric:           struct                  solution information
%RemainVol:        scalar                  remaining parameter space to check
%-------------------------------------------------------------------------
%get number of parameters and variables
%=========================================================================
pn = S(1).Dim;

xn = length(problem.lb);
%=========================================================================
%initialize struct fields, cells, and arrays
%=========================================================================
metric.error = 0; %aggregate error per simplex
metric.intError = 0;
metric.nlps = length(S)*(pn+2);
metric.xError = 0;    %Pangia 3-4-2020
metric.numTooSmallPolys = 0;    %Pangia 5-28-2020
metric.errorList=[]; %individual error per simplex
xOpt = cell(0);
fOpt = cell(0);
CRopt = [];
% tempCap=cap-1;
%=========================================================================
%note 30 is a guessed bound of |H(f*)|;
hessianBound=30;
```

```
43
44 for i = 1:length(S)
45     vertices = S(i).V;
46     CRvol = S(i).volume;
47     M = [vertices,ones(pn+1,1)];
48     X = zeros(xn,pn+1);
49     fval = zeros(1,pn+1);
50
51     %determine optimal solution at each vertex
52     int0 = problem.lb;
53     for j = 1:pn+1
54         [X(:,j),fval(j)] = optimizationProblem(int0,vertices(j,:)',problem);
55     end
56
57     %linear approximations of optimal value function and decision functions
58     %Value Function:          zApprox = fval*(M^-1)[1;t]
59     %Decision Function:       xApprox = X(M^-1)[1;t]
60 %      Minv = M^-1;
61 %      xApprox = (Minv*X')';
62 %      zApprox = (Minv*fval')';
63
64     %1-7-21: note that xApprox=X*M^-1 is the same as solving xApprox*M=X
65     %equivalently, xApprox'=Minv*X' is the same as solving M*xApprox'=X'
66     % which is done by computing xApprox=(M\(X'))'
67     %similarly, compute zApprox=(M\(fval'))'
68     xApprox=(M\(X'))';
69     zApprox=(M\(fval'))';
70
71     %for smaller matrices the calculation time seems to be reduced by
72     %extending the vectors for mldivide (\) and splitting the solution up
73 %      tempAnswer=blkdiag(M,M)\([X';fval']);
74
75     %=========================================================================
```

```
76      %determine maximum error
77      %=========================================================
78      %computes maximum error for zApprox as well as
79      %evaluating f at the linearly interpolated optimizers based on xApprox
80      %see Bemporad 2006 paper for more details
81  %%-------------------------------------------------------------------
82      %Error Type Longest Edge Method (LEM):
83      if strcmp(errorType,'LEM')
84          %extract the list of edgelengths, and the corresponding vertices
85          %from the data stored in S(i):
86          lengthsList=S(i).Data{1};
87          vertexOneList=S(i).Data{2};
88          vertexTwoList=S(i).Data{3};
89
90          %compute the longest edge
91          [longestEdgeLength, longestIndex]=max(lengthsList);
92          vertexOneLongest=vertexOneList(longestIndex);
93          vertexTwoLongest=vertexTwoList(longestIndex);
94
95          canSplit=1;
96          error1=longestEdgeLength^2*hessianBound/8;
97          %if the edge is too long, split
98          if error1 > tol
99              %center of longest edge
100             tHat=((vertices(vertexOneLongest,:)+...
101                 vertices(vertexTwoLongest,:))/2)';
102
103             %split the simplex into two subsimplices; one with the vertex
104             %from vertexOneList replaced by the midpoint, and the other
105             %with the vertex from vertexTwoList replaced by the midpoint;
106             %note that tHat is the midpoint of the longest edge
107             S2(1)=Polyhedron([]);
108             S2(2)=Polyhedron([]);
```

```
109             vertices1=vertices;
110             vertices1(vertexOneLongest,:)=tHat;
111             vertices2=vertices;
112             vertices2(vertexTwoLongest,:)=tHat;
113             S2(1)=Polyhedron(vertices1);
114             S2(2)=Polyhedron(vertices2);
115             %we need to recalculate the
116             %edge lengths, since we don't know which locations we need to
117             %change, or if the vertex is even going to be in the same
118             %location
119             numVertices=size(vertices,1);
120             listLen=numVertices*(numVertices-1)/2;
121             tempLengthsList=zeros(1,listLen);
122             tempVertexOneList=zeros(1,listLen);
123             tempVertexTwoList=zeros(1,listLen);
124             for splitIndex=1:2
125             listPtr=1;
126             for k=1:numVertices-1
127                 for j=k+1:numVertices
128                     tempLengthsList(listPtr)=norm(S2(splitIndex).V(k,:) -...
129                         S2(splitIndex).V(j,:));
130                     tempVertexOneList(listPtr,:)=k;
131                     tempVertexTwoList(listPtr,:)=j;
132                     listPtr=listPtr+1;
133                 end
134             end
135             %attach the edge lengths to the simplex
136             S2(splitIndex).Data={tempLengthsList,tempVertexOneList,
                tempVertexTwoList};
137             %check if S2 is too small
138             if ~S2(splitIndex).isFullDim
139                 canSplit=0;
140             end
```

```
141                 end %splitIndex for
142             end %error if
143         else % end of LEM; on to BOM
144         %Error Type Biconvex Overestimator Method (BOM): max |zApprox(t) - f*(t)|
145         tCenter = mean(vertices)';
146
147         x0 = [xApprox*[tCenter;1];tCenter];
148 % pass the set of solutions as well to compute the biconvex overestimator
149         [xHat,tHat,error1] = computeError(problem,S(i),X,zApprox,x0,zApprox);
150
151         %8-23-2022: to force partitioning
152         %if tHat is infeasible due to fmincon's shenanigans, project it onto
153         %S(i) and then choose the actual next point to be that projection
154         if ~S(i).contains(tHat)
155             tHat=S(i).project(tHat).x;
156         end
157
158         %For nonconvex cases: Check error at midpoint of longest edge
159         %if tHat is a vertex:
160         %1-14-21: consider if ~S(i).contains(tHat) then
161         %tHat=project(S(i),tHat)
162         tM = repmat(tHat',pn+1,1);
163         vertDistance = min(sum(abs(tM-vertices),2));
164
165         if vertDistance <= pn*(1e-5)
166             %6-25-2021: make the split point be the center of the longest edge to
167             %ensure no squat triangles (due to error bounds for interpolation)
168
169             %compute and store the lengths of the vertices
170             numVertices=size(vertices,1);
171             listLen=numVertices*(numVertices-1)/2;
172             lengthsList=zeros(1,listLen);
173             %a list of the row indices of S(k).V corresponding to each edge of the
```

```
174         %simplex
175         vertexOneList=zeros(1,listLen);
176         vertexTwoList=zeros(1,listLen);
177         listPtr=1;
178         for k=1:numVertices-1
179             for j=k+1:numVertices
180                 lengthsList(listPtr)=norm(vertices(k,:)-vertices(j,:));
181                 vertexOneList(listPtr,:)=k;
182                 vertexTwoList(listPtr,:)=j;
183                 listPtr=listPtr+1;
184             end
185         end
186
187
188         %compute the longest edge
189         [longestEdgeLength, longestIndex]=max(lengthsList);
190         vertexOneLongest=vertexOneList(longestIndex);
191         vertexTwoLongest=vertexTwoList(longestIndex);
192
193         %center of longest edge
194         tHat=((vertices(vertexOneLongest,:)+...
195             vertices(vertexTwoLongest,:))/2)';
196
197         %compute the error and then split the simplex into two subsimplices
198         %to try again
199         error1=longestEdgeLength^2*hessianBound/8;
200
201         %split the simplex into two subsimplices; one with the vertex
202         %from vertexOneList replaced by the midpoint, and the other
203         %with the vertex from vertexTwoList replaced by the midpoint;
204         %note that tHat is the midpoint of the longest edge
205         S2(1)=Polyhedron([]);
206         S2(2)=Polyhedron([]);
```

```
207             vertices1=vertices;
208             vertices1(vertexOneLongest,:)=tHat;
209             vertices2=vertices;
210             vertices2(vertexTwoLongest,:)=tHat;
211             S2(1)=Polyhedron(vertices1);
212             S2(2)=Polyhedron(vertices2);
213         else %otherwise the BOM worked
214             S2 = splitSimplex(S(i),tHat);
215         end %adjustment if max point is a vertex
216
217         %%-----------------------------------------------------------------
218         %Error Type 2: max |f(xApprox(t)) - f*(t)|
219         xBar = xApprox*[tHat;1];
220         error2 = abs(problem.objective(xHat,tHat) - ...
221                         problem.objective(xBar,tHat));
222         %%-----------------------------------------------------------------
223         %if the polyhedron S(i) is too small to split (assume it is not)
224         canSplit=~isempty(S2);      %Pangia 5-12-2020
225         end %selection LEM or BOM
226         %=================================================================
227         %Choose which type of error should be used for determining the quality
228         %of the solution:
229         %errorType = error2;
230
231         wahErrorType = error1;
232
233 %Pangia 4-25-2020 commented these out
234 %       disp('************************************************')
235 %       disp(['Largest error at t = ' num2str(tHat')])
236 %       disp(['zApprox error is ' num2str(error1)])
237 %       disp(['f(xApprox) error is ' num2str(error2)])
238 %       disp(['Region Volume is ' num2str(CRvol)])
239 %       disp(['Total Volume Remaining is ' num2str(RemainVol)])
```

```
240 %     disp('************************************************')
241 %     disp(' ')
242
243     if (wahErrorType < tol || ~canSplit || cap <= 0)
244         %Solution is acceptable, save and move to next simplex;
245         %alternatively, solution is unacceptable, but we can't fine-tune it
246         %anymore, or the number of splits has been exceeded
247         xOpt = [xOpt;xApprox];
248         fOpt = [fOpt;zApprox];
249         CRopt = [CRopt;S(i)];
250         metric.errorList= [metric.errorList, metric.error];
251         metric.error = metric.error + wahErrorType;
252         metric.intError = metric.intError + wahErrorType*CRvol;
253         %commented out, but left in if wanted for the future
254 %          metric.xError = metric.xError + error2;     %Pangia 3-4-2020
255         metric.numTooSmallPolys=metric.numTooSmallPolys+~canSplit; %Pangia
                5-29-2020
256         RemainVol = RemainVol - CRvol;
257     else
258         %Error is too large, split simplex and make new solution
259
260         [x2,f2,CR2,metric2,RemainVol] = mpSimplexApproximation(...
261                                     tol,S2,problem,RemainVol,cap-1,errorType);
262         metric.errorList=[metric.errorList, metric2.errorList];
263         metric.error = metric.error + metric2.error;
264         metric.xError= metric.xError + metric2.xError;    %Pangia 3-5-2020
265         metric.intError = metric.intError + metric2.intError;
266         metric.nlps = metric.nlps + metric2.nlps;
267         metric.numTooSmallPolys=metric.numTooSmallPolys+...
268             metric2.numTooSmallPolys;
269         xOpt = [xOpt;x2];
270         fOpt = [fOpt;f2];
271         CRopt = [CRopt;CR2];
```

```
272     end %if
273 end %for
274 end
275
276
277 function [xHat,tHat,error] = computeError(problem,S,X,F,x0,zHat)
278 SA = S.A;
279 Sb = S.b;
280 m = size(SA,1);
281 n = length(problem.lb);
282
283 A = [problem.A, -problem.E;
284       zeros(m,n),SA];
285 b = [problem.b;Sb];
286
287 Aeq=[problem.Aeq, -problem.Eeq];
288 beq=problem.beq;
289
290 lb = [problem.lb;min(S.V)'];
291 ub = [problem.ub;max(S.V)'];
292
293 obj = @(x)nlpObjective(x,n,zHat,problem.objective);
294 if isempty(problem.constraints)
295     const = [];
296 else
297     const = @(x)nlpConstraints(x,n,problem.constraints);
298 end
299
300 options = optimoptions('fmincon','Algorithm','sqp','Display','off');
301 [x1,error1] = fmincon(obj,x0,A,b,Aeq,beq,lb,ub,const,options);
302
303 %compute the difference between the biconvex overestimator and the linear
304 %interpolant
```

```
305 vertices=S.V;
306 vertexNum=size(vertices,1);
307 overObj=@(lambda) F*[vertices'*lambda; 1] - overestSum(vertices,...
308                             problem.objective,X,lambda);
309 lambda0=ones(vertexNum,1)./vertexNum;
310 lbLambda=zeros(vertexNum,1);
311 ubLambda=ones(vertexNum,1);
312 ineqsLambda=SA*vertices';
313 eqLambda=ones(1,vertexNum);
314 [lambda,error2]=fmincon(overObj,lambda0,ineqsLambda,Sb,eqLambda,1,lbLambda,...
315                         ubLambda,[],options);
316
317 %need to also check to make sure that lambda is NOT a vertex, because
318 %that's not helpful; in that case, just set error to be error1, because the
319 %vertex results only when (I think) the objective function is independent
320 %of the parameter, and thus jointly convex
321
322 %all of the weights being less than one means we're far enough from the
323 %vertices that things should work again
324 if all(lambda<0.9999) && abs(error2) > abs(error1)
325     tHat=vertices'*lambda;
326     %xHat is the optimization at tHat
327     [xHat, ~] = optimizationProblem(problem.lb,tHat,problem);
328     error=abs(error2);
329 else
330     error = abs(error1);
331     xHat = x1(1:n);
332     tHat = x1(n+1:end);
333 end
334 %xHat = x(1:n);
335 end
336
337 function f = nlpObjective(u,n,z,func)
```

```
%objective function for mp-nlp with parameters treated as variables
x = u(1:n);
p = u(n+1:end);
%note that this works only for jointly convex functions
f = func(x,p)-z*[p;1];

end

function [c,ceq] = nlpConstraints(u,n,func)
%Nonlinear constraints for mp-nlp with parameters treated as variables
x = u(1:n);
p = u(n+1:end);
[c,ceq] = func(x,p);
end


%overestSum: used to maximize the distance between the biconvex
%overestimator and the linear interpolant
function F=overestSum(vertices, objFunc, X, lambda)
    vertexNum=length(vertices);
    overestMat=lambda*lambda';
    for i=1:vertexNum
        for j=1:vertexNum
            %note that X(:,j) is the solution for vertex j
            overestMat(i,j)=overestMat(i,j)*objFunc(X(:,i),vertices(j,:));
        end
    end
    F=sum(sum(overestMat));
end %end overestSum
```

```
function P = splitSimplex(CR,varargin)
%partitions the simplex CR by connecting each facet of CR with a
%point in CR. Uses specified point if given, otherwise uses the center
%-----------------------------------------------------------------------
```

```
%INPUT                  TYPE                DESCRIPTION
%CR:                    polyhedron          parameter space
%varargin:              cell: vector        point to use to split region
%-------------------------------------------------------------------------
%OUTPUT                 TYPE                DESCRIPTION
%P:                     polyhedron array    parameter space partition
%-------------------------------------------------------------------------
if isempty(varargin)
    center = mean(CR.V)';
else
    center = varargin{1};
end

CR.minVRep;
V = CR.V;
m = size(V,1);
P(m) = Polyhedron([]);

for i = 1:m
    V2 = V;
    V2(i,:) = center';
    Ptemp = Polyhedron(V2);
    if Ptemp.isFullDim && (Ptemp.volume > 0)
        P(i) = Polyhedron(V2);
    end
    clear Ptemp
end
P(P.isEmptySet)=[];
end
```

sec:BASMbench

C.2 Benchmark Functions

The benchmark functions below operate exactly the same way as those in B.2, but with a better written randomization method.

```
function [] = makeConstraints3Param(constraints, numParams,...
                                    numVars, nameIndex)
%create the constraints file that can then be read by fmincon
%-------------------------------------------------------------------------
%INPUT          TYPE                    DESCRIPTION
%constraints:   cell                    quadratic constraints
%numParams:     double                  number of parameters
%numVars:       double                  number of variables
%nameIndex:     double                  which example this is
%-------------------------------------------------------------------------
numCons=length(constraints);
charIdx=num2str(nameIndex);
fName=['nonlcon' num2str(numVars) 'VarsEx' charIdx '.m'];
fileID=fopen(fName,'w');

%note that matlabFunction calls a symbolic vector 'in#' where #represents
%the index at which it is located as soon as the symbolic vector is larger
%than 1x1. For instance, if theta1,theta2\in R and x\in R^2, then
%{theta1,theta2,x} becomes {theta1,theta2,in3} while {x,theta1,theta2}
%becomes {in1,theta1,theta2}
%technically, we should obtain where the vectors are located in the
%constraints, but I'm lazy: the code is forcing it to be 'in3' so that's
%what we'll make it
temp=['function [g,h] = nonlcon' charIdx];
if numParams==3
    firstLine=[temp '(theta1,theta2,theta3,in4)\n'];
elseif numParams==2
    firstLine=[temp '(theta1,theta2,in3)\n'];
else
    firstLine=[temp '(theta1,in2)\n'];
```

```
end

%convert the symbol array x into character representations of
%its individual components, and append them to the firstLine
fprintf(fileID, firstLine);
for i=1:numCons
    %find the first instance of ')' and then delete all parts of the array
    %prior to it; basically the '@(<argin>)'
    temp=char(constraints{i});
    closeParenIdx=strfind(temp,')');
    temp=temp(closeParenIdx(1)+1:end);
    constLine=['g(' num2str(i) ')=' temp ';\n'];
    fprintf(fileID, constLine);
end %for

%modify to accept equality constraints at some point
fprintf(fileID, 'h=[];\n');
fprintf(fileID, 'end');
fclose(fileID);

end %function
```

```
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm\mpASM')
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice')

addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice\randFuncs')

%Random Example (plots)
%three variables
% numX=2;
% numX=3;
```

```
10 % numX=4;
11 numX=5;
12 % xSize=[1,numX];        %row form;
13
14 %MUST BE COLUMN FORM RIGHT NOW
15 xSize=[numX,1];        %col form;
16
17 x=sym('x',xSize);
18 assume(x, 'real');
19
20 %one parameter
21 syms theta1;
22 assume(theta1>=0.1 & theta1<=1.1);
23 % assume(theta2>=0 & theta2<=1);
24
25 %ten examples, with 1 nonlinear constraint each
26 numExamples=10;
27 numNonlinConstraints=1;
28
29 %the parameter space
30
31 Theta = Polyhedron('lb',[0.1],'ub',[1.1]);
32
33 %depth for mp-ASM
34 cap=10;
35
36 %set the precision of the symbols
37 digits(5);
38
39 %set the seed so that there is some semblance of replicability
40 rng default;
41 seedNum=8+numX;
42 rng(seedNum);
```

```
43
44 %set the guaranteed feasible point
45 % bSelect=ones(xSize);       %if x is a row
46 bSelect=ones(fliplr(xSize));         %if x is a col
47
48 %Set the linear constraint Ax + E*theta<=b
49 numLinCon=1;
50 Asize=[numLinCon,numX];
51 Esize=[numLinCon,1];    %1 being for theta1
52 cSize=[1,numLinCon];
53
54 %the theta-picking matrix for the linear matrix
55 RaSize=[numLinCon,numLinCon];
56 RcSize=[numLinCon,numLinCon];
57
58 %the matrices for generating the objective function and the quadratic
59 %constraints
60 %Note Q=P'P
61 Psize=[numX,numX];
62 RpSize=Psize;
63 qSize=[1,numX];
64
65 cQuadSize=[1,1];
66 RcQuadSize=[1,1];
67
68 mpsaEx=cell(1,numExamples);
69 mpsaExTemp=struct;
70
71 %solution fields
72 xSA=cell(1,numExamples);
73 uSA=cell(1,numExamples);
74 vSA=cell(1,numExamples);
75 fSA=cell(1,numExamples);
```

```
76  CRSA=cell(1,numExamples);
77  solnInfoSA=cell(1,numExamples);
78
79  xASM=cell(1,numExamples);
80  fASM=cell(1,numExamples);
81  CRASM=cell(1,numExamples);
82  solnInfoASM=cell(1,numExamples);
83
84  %the constraints
85  gurke=cell(numExamples,numNonlinConstraints);
86  constraintsList=cell(numExamples,numNonlinConstraints);
87
88  %the objectives
89  objList=cell(numExamples,1);
90
91  for i=1:numExamples
92      disp(['Creating Example ' num2str(i) ' of ' num2str(numExamples)]);
93      mpsaExTemp.A=floor(randMat(Asize,0,1)*1000)/1000;
94      mpsaExTemp.E=floor(randMat(Esize)*rand01([1,1],0.5)*1000)/1000;
95      %[1,1] being for theta1
96      mpsaExTemp.b=mpsaExTemp.A*bSelect';
97      mpsaExTemp.Aeq = [];
98      mpsaExTemp.Eeq = [];
99      mpsaExTemp.beq = [];
100
101     %unrelaxed nonlinear constraints
102     mpsaExTemp.baseConstraints=[];
103
104     %make the objective function
105     %Q=P'(theta)*P(theta)
106     %P(theta)=P1+P2*Rp*theta
107     P1=floor(randMat(Psize,0,1)*1000)/1000;
108     P2=floor(randMat(Psize,0,1)*1000)/1000;
```

```
109
110     Rp=floor(rand01(RpSize,0.5)*1000)/1000;
111
112     %q'(theta)*x
113     q1=floor(randMat(qSize,0,1)*1000)/1000;
114     q2=floor(randMat(qSize,0,1)*1000)/1000;
115     Rq=floor(rand01(RpSize,0.5)*1000)/1000;
116
117     %vpa: variable-precision-arithmetic
118     P=vpa(P1+P2*Rp*theta1);
119     Q=expand(P'*P);
120     q=vpa(q1+q2*Rq*theta1);
121
122
123 %       obj=x*Q*x'+q*x';      %if x is a row
124     obj=vpa(x'*Q*x+q*x);      %if x is a col
125     objList{i}=vpa(x'*Q*x+q*x);
126     obj=matlabFunction(obj,'Vars',{theta1,x});
127
128     %objective function; inner variables must be different than syms
129     mpsaExTemp.objective=@(varX,varTheta) ...
130     obj(varTheta(1),varX);
131
132     for j=1:numNonlinConstraints
133         %Q=P'(theta)*P(theta)
134         %P(theta)=P1+P2*Rp*theta
135         P1=floor(randMat(Psize,0,1)*1000)/1000;
136         P2=floor(randMat(Psize,0,1)*1000)/1000;
137         Rp=floor(rand01(RpSize,0.5)*1000)/1000;
138
139         %q'(theta)*x
140         q1=floor(randMat(qSize,0,1)*1000)/1000;
141         q2=floor(randMat(qSize,0,1)*1000)/1000;
```

```
142         Rq=floor(rand01(RpSize,0.5)*1000)/1000;
143
144         %c(theta)
145         cQuad1=floor(randMat(cQuadSize,0,1)*1000)/1000;
146         cQuad2=floor(randMat(cQuadSize,0,1)*1000)/1000;
147         RcQuad=floor(rand01(cQuadSize,0.5)*1000)/1000;
148
149         P=vpa(P1+P2*Rp*theta1);
150         Q=expand(P'*P);
151         q=vpa(q1+q2*Rq*theta1);
152         cQuad=vpa(cQuad1+cQuad2*RcQuad*theta1);
153
154
155         %b=bSelect'*Q(theta)*bSelect+q'(theta)*bSelect+cQuad(theta)
156         b=bSelect*Q*bSelect'+q*bSelect'+cQuad;
157
158         %generate the nonlinear constraints for mp-SA
159 %           gurke{j}=x*Q*x'+q*x'+cQuad-b;          %if x is a row
160         gurke{i,j}=vpa(x'*Q*x+q*x+cQuad-b);     %if x is a col
161         constraintsList{i,j}=gurke{i,j};
162         gurke{i,j}=matlabFunction(gurke{i,j},'Vars',{theta1,x});
163         %constraints to be relaxed; inner variables must be different than
164         %syms
165         mpsaExTemp.relConIneq{j}=@(varX,varTheta) ...
166             gurke{i,j}(varTheta(1),varX);
167     end
168     %create the nonlcon constraint for mp-ASM
169     makeConstraints(gurke(i),0,numX,i);
170
171     mpsaExTemp.relConEq=cell(0);
172
173     %lower and upper bounds for x
174     %WARNING: need to be columns for mpASM
```

```
175     boundDim=xSize;
176
177     mpsaExTemp.lb = zeros(boundDim);
178     mpsaExTemp.ub = inf*ones(boundDim);
179
180     mpsaEx{i}=mpsaExTemp;
181 end
182
183 %compute mp-ASM
184 for i=1:numExamples
185     %compute the example with the proper constraints
186     filename=['nonlcon' num2str(numX) 'VarsEx' num2str(i) '.m'];
187     copyfile(filename, 'nonlcon.m');
188     mpsaEx{i}.constraints=@(x,theta) nonlcon(theta(1),x);
189
190     %run the same example to see how straight mpASM compares
191 %       [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
192 %           mpASM(0.01,Theta,mpsaEx{i},cap,'BOM');
193     [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
194         mpASM(0.01,Theta,mpsaEx{i},cap,'LEM');
195
196     %after computing, draw the graphs of the functions that have poor error
197     %values
198     if(1)
199         %make a grid pattern of the actual solution function
200         numPoints=21;
201         thetaTemp=linspace(0,1,numPoints);
202
203         %make cells of the solutions;
204         xGrid=cell(numX,1);
205         fGrid=zeros(numPoints,1);
206         x0=mpsaEx{i}.lb;
207
```

```
208         options = optimoptions('fmincon',...
209             'Algorithm','sqp','Display','Off');
210         for idx=1:length(thetaTemp)
211             A=mpsaEx{i}.A;
212             b=mpsaEx{i}.b+mpsaEx{i}.E*thetaTemp(idx);
213             obj=@(x) mpsaEx{i}.objective(x,thetaTemp(idx));
214             const=@(x) mpsaEx{i}.constraints(x,thetaTemp(idx));
215             [tempX,fGrid(idx)]=fmincon(obj,x0,A,b,[],[],...
216                 mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
217             for idx2=1:numX
218                 xGrid{idx2}(idx)=tempX(idx2);
219             end
220         end %for
221
222         gurkeFig=figure;
223         plot(thetaTemp,fGrid,'o');
224         xlabel('\theta_1');
225         ylabel('f');
226         plotParametricFunction(fASM{i},CRASM{i});
227         title(['mpASM ' num2str(numX) ' Vars Optimal Value Function' ...
228             ' for Example' num2str(i)]);
229         filename=['ASM' num2str(numX) 'VarsVal' num2str(i) '.fig'];
230         saveas(gurkeFig,filename);
231     end%if
232 end
233
234
235
236 %collect the data
237 errorASM=zeros(1,numExamples);
238 xErrorASM=zeros(1,numExamples);
239 timeASM=zeros(1,numExamples);
240 numPolys=zeros(1,numExamples);
```

```
241
242 %we want the error to be average error, not total error: total error makes
243 %no sense given the number of polytopes that we could be encountering over
244 %higher dimensions
245 for i=1:numExamples
246     numPolys(i)=length(CRASM{i});
247     errorASM(i)=solnInfoASM{i}.error/numPolys(i);
248 %     xErrorASM(i)=solnInfoASM{i}.xError;
249     timeASM(i)=solnInfoASM{i}.time;
250 end
251
252
253
254 errorStats=struct;
255 errorStats.meanVal=mean(errorASM);
256 errorStats.medianVal=median(errorASM);
257 errorStats.stdDevVal=std(errorASM);
258
259 xErrStats=struct;
260 xErrStats.mean=mean(xErrorASM);
261 xErrStats.median=median(xErrorASM);
262 xErrStats.stdDev=std(xErrorASM);
263
264 timeStats=struct;
265 timeStats.mean=mean(timeASM);
266 timeStats.median=median(timeASM);
267 timeStats.stdDev=std(timeASM);
268
269 polyStats=struct;
270 polyStats.mean=mean(numPolys);
271 polyStats.median=median(numPolys);
272 polyStats.stdDev=std(numPolys);
```

```
1 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
```

```
    Matlab codes\mp ASM algorithm\mpASM')
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice')

addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice\randFuncs')

%Random Example (plots)
%three variables
% numX=2;
% numX=3;
% numX=4;
numX=5;
% xSize=[1,numX];       %row form;

%MUST BE COLUMN FORM RIGHT NOW
xSize=[numX,1];       %col form;

x=sym('x',xSize);
assume(x, 'real');

%two parameters
syms theta1 theta2;
assume(theta1>=0.1 & theta1<=1.1);
assume(theta2>=0.1 & theta2<=1.1);

%ten examples, with 1 nonlinear constraint each
%3-8-22: sixth example is bad; cut it
numExamples=11;
% numExamples=14;  %14 for 4 vars, going from 0.2 to 0.6
% numExamples=13;  %13 for 5 vars, going from 0.2 to 0.6
numNonlinConstraints=1;

```

```
%the parameter space for 4 and 5 variables
% theta1Lower=0.2;
% theta2Lower=0.2;
% theta1Upper=0.6;
% theta2Upper=0.6;

%the parameter space for 2 and 3 variables
theta1Lower=0.1;
theta2Lower=0.1;
theta1Upper=1.1;
theta2Upper=1.1;
Theta = Polyhedron('lb',[theta1Lower,theta2Lower],...
    'ub',[theta1Upper,theta2Upper]);

%depth for mp-ASM
cap=11;

%number of dual updates for mp-SA
maxIter=10;

%initial guess for dual variables: v does not exist
%Note u0={zeros(1,(1+numParams))};
u0={zeros(1,3)};
v0={};

%set the precision of the symbols
digits(5);

%set the seed so that there is some semblance of replicability
% seedNum=100; %20; doesn't work: 10
rng default;
seedNum=8+numX;
rng(seedNum);
```

```
65
66 %set the guaranteed feasible point
67 % bSelect=ones(xSize);       %if x is a row
68 bSelect=ones(fliplr(xSize));         %if x is a col
69
70 %Set the linear constraint Ax + E*theta<=b
71 numLinCon=1;
72 Asize=[numLinCon,numX];
73 Esize=[numLinCon,2];    %2 being for theta1 and theta2
74 cSize=[1,numLinCon];
75
76 %the theta-picking matrix for the linear matrix
77 RaSize=[numLinCon,numLinCon];
78 RcSize=[numLinCon,numLinCon];
79
80 %the matrices for generating the objective function and the quadratic
81 %constraints
82 %Note Q=P'P
83 Psize=[numX,numX];
84 RpSize=Psize;
85 qSize=[1,numX];
86
87 cQuadSize=[1,1];
88 RcQuadSize=[1,1];
89
90 mpsaEx=cell(1,numExamples);
91 mpsaExTemp=struct;
92
93 %solution fields
94 xSA=cell(1,numExamples);
95 uSA=cell(1,numExamples);
96 vSA=cell(1,numExamples);
97 fSA=cell(1,numExamples);
```

```
98  CRSA=cell(1,numExamples);
99  solnInfoSA=cell(1,numExamples);
100
101 xASM=cell(1,numExamples);
102 fASM=cell(1,numExamples);
103 CRASM=cell(1,numExamples);
104 solnInfoASM=cell(1,numExamples);
105
106 %the constraints
107 gurke=cell(numExamples,numNonlinConstraints);
108 constraintsList=cell(numExamples,numNonlinConstraints);
109
110 %the objectives
111 objList=cell(numExamples,1);
112
113 for i=1:numExamples
114     disp(['Creating Example ' num2str(i) ' of ' num2str(numExamples)]);
115     mpsaExTemp.A=floor(randMat(Asize,0,1)*1000)/1000;
116     %make a matrix of numbers ranging from 0 to 1, to determine whether
117     %theta_i is present in the linear constraints; less than 0.5
118     %means not present
119     thetaSelect=rand(Esize);
120     mpsaExTemp.E=floor(randMat(Esize)*1000)/1000;
121     mpsaExTemp.E(thetaSelect<0.5)=0;
122
123     mpsaExTemp.b=mpsaExTemp.A*bSelect';
124     mpsaExTemp.Aeq = [];
125     mpsaExTemp.Eeq = [];
126     mpsaExTemp.beq = [];
127
128     %unrelaxed nonlinear constraints
129     mpsaExTemp.baseConstraints=[];
130
```

```
131     %make the objective function: f(x;theta)=x^TQ(theta)x+q^T(theta)x
132
133     %Q=P'(theta)*P(theta)
134     %P(theta)=P1+P2(theta)
135     P1=floor(randMat(Psize,0,1)*1000)/1000;
136     P2=sym(floor(randMat(Psize,0,1)*1000)/1000);
137
138     %put theta1, theta2, or no parameter into the quadratic form of the
139     %objective function
140     thetaSelect=rand(Psize);
141     noneHere=thetaSelect<0.33;
142     theta1Here=thetaSelect>=0.33 & thetaSelect< 0.67;
143     theta2Here=thetaSelect>=0.67;
144     P2(noneHere)=0;
145     P2(theta1Here)=P2(theta1Here).*theta1;
146     P2(theta2Here)=P2(theta2Here).*theta2;
147     P=vpa(P1+P2);
148     Q=expand(P'*P);
149
150     %q'(theta)*x
151     q1=floor(randMat(qSize,0,1)*1000)/1000;
152     q2=sym(floor(randMat(qSize,0,1)*1000)/1000);
153
154     %put theta1, theta2, or no parameter into the linear portion of the
155     %objective function
156     thetaSelect=rand(qSize);
157     noneHere=thetaSelect<0.33;
158     theta1Here=thetaSelect>=0.33 & thetaSelect< 0.67;
159     theta2Here=thetaSelect>=0.67;
160     q2(noneHere)=0;
161     q2(theta1Here)=q2(theta1Here).*theta1;
162     q2(theta2Here)=q2(theta2Here).*theta2;
163     q=vpa(q1+q2);
```

```
164
165 %       obj=x*Q*x'+q*x';      %if x is a row
166       obj=vpa(x'*Q*x+q*x);      %if x is a col
167       objList{i}=vpa(x'*Q*x+q*x);
168       obj=matlabFunction(obj,'Vars',{theta1,theta2,x});
169
170       %objective function; inner variables must be different than syms
171       mpsaExTemp.objective=@(varX,varTheta) ...
172       obj(varTheta(1),varTheta(2),varX);
173
174       for j=1:numNonlinConstraints
175           %Q=P'(theta)*P(theta)
176           %P(theta)=P1+P2*Rp*theta
177           P1=floor(randMat(Psize,0,1)*1000)/1000;
178           P2=sym(floor(randMat(Psize,0,1)*1000)/1000);
179           %put theta1, theta2, or no parameter into the quadratic form of the
180           %constraint function
181           thetaSelect=rand(Psize);
182           noneHere=thetaSelect<0.33;
183           theta1Here=thetaSelect>=0.33 & thetaSelect< 0.67;
184           theta2Here=thetaSelect>=0.67;
185           P2(noneHere)=0;
186           P2(theta1Here)=P2(theta1Here).*theta1;
187           P2(theta2Here)=P2(theta2Here).*theta2;
188           P=vpa(P1+P2);
189           Q=expand(P'*P);
190
191           %q'(theta)*x
192           q1=floor(randMat(qSize,0,1)*1000)/1000;
193           q2=sym(floor(randMat(qSize,0,1)*1000)/1000);
194           %put theta1, theta2, or no parameter into the linear portion of the
195           %constraint function
196           thetaSelect=rand(qSize);
```

```
197         noneHere=thetaSelect<0.33;
198         theta1Here=thetaSelect>=0.33 & thetaSelect< 0.67;
199         theta2Here=thetaSelect>=0.67;
200         q2(noneHere)=0;
201         q2(theta1Here)=q2(theta1Here).*theta1;
202         q2(theta2Here)=q2(theta2Here).*theta2;
203         q=vpa(q1+q2);
204
205         %c(theta)
206         cQuad1=floor(randMat(cQuadSize,0,1)*1000)/1000;
207         cQuad2=sym(floor(randMat(cQuadSize,0,1)*1000)/1000);
208         %put theta1, theta2, or no parameter into the constant portion of
209         %the objective function
210         thetaSelect=rand(cQuadSize);
211         noneHere=thetaSelect<0.33;
212         theta1Here=thetaSelect>=0.33 & thetaSelect< 0.67;
213         theta2Here=thetaSelect>=0.67;
214         cQuad2(noneHere)=0;
215         cQuad2(theta1Here)=cQuad2(theta1Here).*theta1;
216         cQuad2(theta2Here)=cQuad2(theta2Here).*theta2;
217         cQuad=vpa(cQuad1+cQuad2);
218
219         %b=bSelect'*Q(theta)*bSelect+q'(theta)*bSelect+cQuad(theta)
220         b=bSelect*Q*bSelect'+q*bSelect'+cQuad;
221
222         %generate the nonlinear constraints for mp-SA
223 %           gurke{j}=x*Q*x'+q*x'+cQuad-b;         %if x is a row
224         gurke{i,j}=vpa(x'*Q*x+q*x+cQuad-b);     %if x is a col
225         constraintsList{i,j}=gurke{i,j};
226         gurke{i,j}=matlabFunction(gurke{i,j},'Vars',{theta1,theta2,x});
227         %constraints to be relaxed; inner variables must be different than
228         %syms
229         mpsaExTemp.relConIneq{j}=@(varX,varTheta) ...
```

```
230                 gurke{i,j}(varTheta(1),varTheta(2),varX);
231         end
232         %create the nonlcon constraint for mp-ASM
233         makeConstraints(gurke(i),true,numX,i);
234
235         mpsaExTemp.relConEq=cell(0);
236
237         %lower and upper bounds for x
238         %WARNING: need to be columns for mpASM
239         boundDim=xSize;
240         mpsaExTemp.lb = zeros(boundDim);
241         mpsaExTemp.ub = 10^5*ones(boundDim);
242
243         mpsaEx{i}=mpsaExTemp;
244 end
245
246 %delete the sixth example
247 mpsaEx(6)=[];
248 numExamples=numExamples-1;
249
250 %compute mp-ASM
251 for i=1:numExamples
252         %compute the example with the proper constraints
253         disp(['Solving Example ' num2str(i) ' of ' num2str(numExamples)]);
254         filename=['nonlcon' num2str(numX) 'VarsEx' num2str(i) '.m'];
255         copyfile(filename, 'nonlcon.m');
256         mpsaEx{i}.constraints=@(x,theta) nonlcon(theta(1),theta(2),x);
257
258         %run the same example to see how straight mpASM compares
259         [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
260             mpASM(0.01,Theta,mpsaEx{i},cap,'LEM');
261 %       [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
262 %           mpASM(0.01,Theta,mpsaEx{i},cap,'BOM');
```

```
263 end
264
265 %store how far the example stretches: 0.5 error is big for an example going
266 %from 1 to 3, but small for an example from 100 to 300
267 fRanges=zeros(1,numExamples);
268 %display the examples
269 for i=1:numExamples
270     %after computing, draw the graphs of the functions that have poor error
271     %values
272     disp(['Plotting Example ' num2str(i) ' of ' num2str(numExamples)]);
273     if(1)
274         %make a grid pattern of the actual solution function
275         numPoints=21;
276         thetaTemp=zeros(2,numPoints);
277         thetaTemp(1,:)=linspace(theta1Lower,theta1Upper,numPoints);
278         thetaTemp(2,:)=linspace(theta2Lower,theta2Upper,numPoints);
279
280         %make cells of the solutions; they're square for the surf command
281         xGrid=cell(numX,1);
282         for idx=1:numX
283             xGrid{idx}=zeros(numPoints,numPoints);
284         end
285         uGrid=cell(numNonlinConstraints,1);
286         for idx=1:numNonlinConstraints
287             uGrid{idx}=zeros(numPoints,numPoints);
288         end
289
290         fGrid=zeros(numPoints,numPoints);
291         x0=mpsaEx{i}.lb;
292
293         options = optimoptions('fmincon',...
294             'Algorithm','sqp','Display','Off');
295 %               'Algorithm','sqp','Display','Final');
```

```
296         for idx1=1:length(thetaTemp)
297             for idx2=1:length(thetaTemp)
298                 A=mpsaEx{i}.A;
299                 %WARNING: make this work right; check number of parameters
300                 b=mpsaEx{i}.b+mpsaEx{i}.E*[thetaTemp(1,idx1);...
301                     thetaTemp(2,idx2)];
302 %                   b=mpsaEx{i}.b+mpsaEx{i}.E*[thetaTemp(1,idx1)];
303                 objTemp=@(x) mpsaEx{i}.objective(x,[thetaTemp(1,idx1),...
304                     thetaTemp(2,idx2)]);
305                 const=@(x) mpsaEx{i}.constraints(x,[thetaTemp(1,idx1),...
306                     thetaTemp(2,idx2)]);
307
308                 %hth column, fill the rows
309                 [temp,fGrid(idx2,idx1),~,~,tempDual,~,~]=fmincon(objTemp,...
310                     x0,A,b,[],[],mpsaEx{i}.lb,mpsaEx{i}.ub,const,options);
311                 for idx=1:numX
312                     xGrid{idx}(idx2,idx1)=temp(idx);
313                 end
314                 for idx=1:numNonlinConstraints
315                     uGrid{idx}(idx2,idx1)=tempDual.ineqnonlin(idx);
316                 end
317
318             end %for
319         end
320         [thetaX, thetaY]=meshgrid(thetaTemp(1,:),thetaTemp(2,:));
321         gurkeFig=figure;
322         surf(thetaX,thetaY,fGrid);
323         xlabel('\theta_1');
324         ylabel('\theta_2');
325         plotParametricFunction(fASM{i},CRASM{i});
326         title(['mpASM ' num2str(numX) ' Vars Optimal Value Function' ...
327             ' for Example' num2str(i)]);
328         filename=['ASM' num2str(numX) 'VarsVal' num2str(i) '.fig'];
```

```
329             saveas(gurkeFig,filename);
330
331             fRanges(i)=abs(max(fGrid(:))-min(fGrid(:)));
332         end%if
333 end
334
335
336 %collect the data
337 rawErrASM=zeros(1,numExamples);
338 errorASM=zeros(1,numExamples);
339 xErrorASM=zeros(1,numExamples);
340 timeASM=zeros(1,numExamples);
341 numPolys=zeros(1,numExamples);
342
343
344 %we want the error to be average error, not total error: total error makes
345 %no sense given the number of polytopes that we could be encountering over
346 %higher dimensions
347 for i=1:numExamples
348         numPolys(i)=length(CRASM{i});
349         errorASM(i)=solnInfoASM{i}.error/numPolys(i);
350         xErrorASM(i)=solnInfoASM{i}.xError;
351         timeASM(i)=solnInfoASM{i}.time;
352 end
353
354 %for five variables: omit 7, 9, and 13
355 % errorASM(13)=[];
356 % errorASM(9)=[];
357 % errorASM(7)=[];
358 % xErrorASM(13)=[];
359 % xErrorASM(9)=[];
360 % xErrorASM(7)=[];
361 % timeASM(13)=[];
```

```
362 % timeASM(9)=[];
363 % timeASM(7)=[];
364
365 errorStats=struct;
366 errorStats.meanVal=mean(errorASM);
367 errorStats.medianVal=median(errorASM);
368 errorStats.stdDevVal=std(errorASM);
369
370 timeStats=struct;
371 timeStats.mean=mean(timeASM);
372 timeStats.median=median(timeASM);
373 timeStats.stdDev=std(timeASM);
374
375 polyStats=struct;
376 polyStats.mean=mean(numPolys);
377 polyStats.median=median(numPolys);
378 polyStats.stdDev=std(numPolys);
```

```
1  addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
       Matlab codes\mp ASM algorithm\mpASM')
2  addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
       MPT Practice\mpSAPractice')
3
4  addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
       MPT Practice\mpSAPractice\randFuncs')
5
6  %Random Example (plots)
7  % numX=2;
8  numX=3;
9  % numX=4;
10 % numX=5;
11 % xSize=[1,numX];       %row form;
12
13 %MUST BE COLUMN FORM RIGHT NOW
```

```
14 xSize=[numX,1];       %col form;
15
16 x=sym('x',xSize);
17 assume(x, 'real');
18
19 %two parameters
20 syms theta1 theta2 theta3;
21 assume(theta1>=0.1 & theta1<=1.1);
22 assume(theta2>=0.1 & theta2<=1.1);
23 assume(theta3>=0.1 & theta3<=1.1);
24
25 %ten examples, with 1 nonlinear constraint each
26 %3-7-22: first example is bad; cut it
27 numExamples=11;  %15 for 2 vars; and 3 vars, going from 0.1 to 0.9
28 % numExamples=14;  %14 for 4 vars, going from 0.2 to 0.6
29 % numExamples=13;  %13 for 5 vars, going from 0.2 to 0.6
30 numNonlinConstraints=1;
31
32 %the parameter space for 4 and 5 variables
33 % theta1Lower=0.2;
34 % theta2Lower=0.2;
35 % theta1Upper=0.6;
36 % theta2Upper=0.6;
37
38 %the parameter space for 2 and 3 variables
39 theta1Lower=0.1;
40 theta2Lower=0.1;
41 theta3Lower=0.1;
42 theta1Upper=1.1;
43 theta2Upper=1.1;
44 theta3Upper=1.1;
45 Theta = Polyhedron('lb',[theta1Lower,theta2Lower,theta3Lower],...
46     'ub',[theta1Upper,theta2Upper,theta3Upper]);
```

```
47
48 %depth for mp-ASM; 12 for LEM, 10 for BOM
49 cap=12;
50
51 %number of dual updates for mp-SA
52 maxIter=10;
53
54 %initial guess for dual variables: v does not exist
55 %Note u0={zeros(1,(1+numParams))};
56 u0={zeros(1,4)};
57 v0={};
58
59 %set the precision of the symbols
60 digits(5);
61
62 %set the seed so that there is some semblance of replicability
63 % seedNum=100; %20; doesn't work: 10
64 rng default;
65 seedNum=8+numX;
66 rng(seedNum);
67
68 %set the guaranteed feasible point
69 % bSelect=ones(xSize);       %if x is a row
70 bSelect=ones(fliplr(xSize));          %if x is a col
71
72 %Set the linear constraint Ax + E*theta<=b
73 numLinCon=1;
74 Asize=[numLinCon,numX];
75 Esize=[numLinCon,3];    %3 being for theta1 theta2 and theta3
76 cSize=[1,numLinCon];
77
78 %the theta-picking matrix for the linear matrix
79 RaSize=[numLinCon,numLinCon];
```

```
80  RcSize=[numLinCon,numLinCon];
81
82  %the matrices for generating the objective function and the quadratic
83  %constraints
84  %Note Q=P'P
85  Psize=[numX,numX];
86  RpSize=Psize;
87  qSize=[1,numX];
88
89  cQuadSize=[1,1];
90  RcQuadSize=[1,1];
91
92  mpsaEx=cell(1,numExamples);
93  mpsaExTemp=struct;
94
95  %solution fields
96  xSA=cell(1,numExamples);
97  uSA=cell(1,numExamples);
98  vSA=cell(1,numExamples);
99  fSA=cell(1,numExamples);
100 CRSA=cell(1,numExamples);
101 solnInfoSA=cell(1,numExamples);
102
103 xASM=cell(1,numExamples);
104 fASM=cell(1,numExamples);
105 CRASM=cell(1,numExamples);
106 solnInfoASM=cell(1,numExamples);
107
108 %the constraints
109 gurke=cell(numExamples,numNonlinConstraints);
110 constraintsList=cell(numExamples,numNonlinConstraints);
111
112 %the objectives
```

```
113 objList=cell(numExamples,1);
114
115 for i=1:numExamples
116     disp(['Creating Example ' num2str(i) ' of ' num2str(numExamples)]);
117     mpsaExTemp.A=floor(randMat(Asize,0,1)*1000)/1000;
118     %make a matrix of numbers ranging from 0 to 1, to determine whether
119     %theta_i is present in the linear constraints; less than 0.5
120     %means not present
121     thetaSelect=rand(Esize);
122     mpsaExTemp.E=floor(randMat(Esize)*1000)/1000;
123     mpsaExTemp.E(thetaSelect<0.5)=0;
124
125     mpsaExTemp.b=mpsaExTemp.A*bSelect';
126     mpsaExTemp.Aeq = [];
127     mpsaExTemp.Eeq = [];
128     mpsaExTemp.beq = [];
129
130     %unrelaxed nonlinear constraints
131     mpsaExTemp.baseConstraints=[];
132
133     %make the objective function: f(x;theta)=x^TQ(theta)x+q^T(theta)x
134
135     %Q=P'(theta)*P(theta)
136     %P(theta)=P1+P2(theta)
137     P1=floor(randMat(Psize,0,1)*1000)/1000;
138     P2=sym(floor(randMat(Psize,0,1)*1000)/1000);
139
140     %put theta1, theta2, or no parameter into the quadratic form of the
141     %objective function
142     thetaSelect=rand(Psize);
143     noneHere=thetaSelect<0.25;
144     theta1Here=thetaSelect>=0.25 & thetaSelect< 0.5;
145     theta2Here=thetaSelect>=0.5 & thetaSelect< 0.75;
```

```
146     theta3Here=thetaSelect>=0.75;
147     P2(noneHere)=0;
148     P2(theta1Here)=P2(theta1Here).*theta1;
149     P2(theta2Here)=P2(theta2Here).*theta2;
150     P2(theta3Here)=P2(theta3Here).*theta3;
151     P=vpa(P1+P2);
152     Q=expand(P'*P);
153
154     %q'(theta)*x
155     q1=floor(randMat(qSize,0,1)*1000)/1000;
156     q2=sym(floor(randMat(qSize,0,1)*1000)/1000);
157
158     %put theta1, theta2, or no parameter into the linear portion of the
159     %objective function
160     thetaSelect=rand(qSize);
161     noneHere=thetaSelect<0.25;
162     theta1Here=thetaSelect>=0.25 & thetaSelect< 0.5;
163     theta2Here=thetaSelect>=0.5 & thetaSelect<0.75;
164     theta3Here=thetaSelect>0.75;
165     q2(noneHere)=0;
166     q2(theta1Here)=q2(theta1Here).*theta1;
167     q2(theta2Here)=q2(theta2Here).*theta2;
168     q2(theta3Here)=q2(theta3Here).*theta3;
169     q=vpa(q1+q2);
170
171 %     obj=x*Q*x'+q*x';     %if x is a row
172     obj=vpa(x'*Q*x+q*x);     %if x is a col
173     objList{i}=vpa(x'*Q*x+q*x);
174     obj=matlabFunction(obj,'Vars',{theta1,theta2,theta3,x});
175
176     %objective function; inner variables must be different than syms
177     mpsaExTemp.objective=@(varX,varTheta) ...
178     obj(varTheta(1),varTheta(2),varTheta(3),varX);
```

```
179
180     for j=1:numNonlinConstraints
181         %Q=P'(theta)*P(theta)
182         %P(theta)=P1+P2*Rp*theta
183         P1=floor(randMat(Psize,0,1)*1000)/1000;
184         P2=sym(floor(randMat(Psize,0,1)*1000)/1000);
185         %put theta1, theta2, or no parameter into the quadratic form of the
186         %constraint function
187         thetaSelect=rand(Psize);
188         noneHere=thetaSelect<0.25;
189         theta1Here=thetaSelect>=0.25 & thetaSelect< 0.5;
190         theta2Here=thetaSelect>=0.5 & thetaSelect<0.75;
191         theta3Here=thetaSelect>=0.75;
192         P2(noneHere)=0;
193         P2(theta1Here)=P2(theta1Here).*theta1;
194         P2(theta2Here)=P2(theta2Here).*theta2;
195         P2(theta3Here)=P2(theta3Here).*theta3;
196         P=vpa(P1+P2);
197         Q=expand(P'*P);
198
199         %q'(theta)*x
200         q1=floor(randMat(qSize,0,1)*1000)/1000;
201         q2=sym(floor(randMat(qSize,0,1)*1000)/1000);
202         %put theta1, theta2, or no parameter into the linear portion of the
203         %constraint function
204         thetaSelect=rand(qSize);
205         noneHere=thetaSelect<0.25;
206         theta1Here=thetaSelect>=0.25 & thetaSelect< 0.5;
207         theta2Here=thetaSelect>=0.5 & thetaSelect< 0.75;
208         theta3Here=thetaSelect>= 0.75;
209         q2(noneHere)=0;
210         q2(theta1Here)=q2(theta1Here).*theta1;
211         q2(theta2Here)=q2(theta2Here).*theta2;
```

```
212         q2(theta3Here)=q2(theta3Here).*theta3;
213         q=vpa(q1+q2);
214
215         %c(theta)
216         cQuad1=floor(randMat(cQuadSize,0,1)*1000)/1000;
217         cQuad2=sym(floor(randMat(cQuadSize,0,1)*1000)/1000);
218         %put theta1, theta2, or no parameter into the constant portion of
219         %the objective function
220         thetaSelect=rand(cQuadSize);
221         noneHere=thetaSelect<0.25;
222         theta1Here=thetaSelect>=0.25 & thetaSelect< 0.5;
223         theta2Here=thetaSelect>=0.5 & thetaSelect< 0.75;
224         theta3Here=thetaSelect>= 0.75;
225         cQuad2(noneHere)=0;
226         cQuad2(theta1Here)=cQuad2(theta1Here).*theta1;
227         cQuad2(theta2Here)=cQuad2(theta2Here).*theta2;
228         cQuad2(theta3Here)=cQuad2(theta3Here).*theta3;
229         cQuad=vpa(cQuad1+cQuad2);
230
231         %b=bSelect'*Q(theta)*bSelect+q'(theta)*bSelect+cQuad(theta)
232         b=bSelect*Q*bSelect'+q*bSelect'+cQuad;
233
234         %generate the nonlinear constraints for mp-SA
235 %           gurke{j}=x*Q*x'+q*x'+cQuad-b;          %if x is a row
236         gurke{i,j}=vpa(x'*Q*x+q*x+cQuad-b);      %if x is a col
237         constraintsList{i,j}=gurke{i,j};
238         gurke{i,j}=matlabFunction(gurke{i,j},'Vars',{theta1,theta2,...
239                                 theta3,x});
240         %constraints to be relaxed; inner variables must be different than
241         %syms
242         mpsaExTemp.relConIneq{j}=@(varX,varTheta) ...
243             gurke{i,j}(varTheta(1),varTheta(2),varTheta(3),varX);
244     end
```

```
245     %create the nonlcon constraint for mp-ASM
246     makeConstraints3Param(gurke(i),3,numX,i);
247
248     mpsaExTemp.relConEq=cell(0);
249
250     %lower and upper bounds for x
251     %WARNING: need to be columns for mpASM
252     boundDim=xSize;
253     mpsaExTemp.lb = zeros(boundDim);
254 %       mpsaExTemp.lb = -5*ones(boundDim);
255     mpsaExTemp.ub = 10^5*ones(boundDim);
256
257
258     mpsaEx{i}=mpsaExTemp;
259 end
260
261 %delete the first example
262 mpsaEx(1)=[];
263 numExamples=numExamples-1;
264
265 %compute mp-ASM
266 for i=1:numExamples
267     %compute the example with the proper constraints
268     disp(['Solving Example ' num2str(i) ' of ' num2str(numExamples)]);
269     filename=['nonlcon' num2str(numX) 'VarsEx' num2str(i) '.m'];
270     copyfile(filename, 'nonlcon.m');
271     mpsaEx{i}.constraints=@(x,theta) nonlcon(theta(1),theta(2),theta(3),x);
272
273     %run the same example to see how straight mpASM compares
274     [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
275         mpASM(0.01,Theta,mpsaEx{i},cap,'LEM');
276 %       [xASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
277 %           mpASM(0.01,Theta,mpsaEx{i},cap,'BOM');
```

```
278 end
279
280 %store how far the example stretches: 0.5 error is big for an example going
281 %from 1 to 3, but small for an example from 100 to 300
282 fRanges=zeros(1,numExamples);
283
284
285 %collect the data
286 rawErrASM=zeros(1,numExamples);
287 errorASM=zeros(1,numExamples);
288 xErrorASM=zeros(1,numExamples);
289 timeASM=zeros(1,numExamples);
290 numPolys=zeros(1,numExamples);
291
292
293 %we want the error to be average error, not total error: total error makes
294 %no sense given the number of polytopes that we could be encountering over
295 %higher dimensions
296 for i=1:numExamples
297     numPolys(i)=length(CRASM{i});
298     errorASM(i)=solnInfoASM{i}.error/numPolys(i);
299     xErrorASM(i)=solnInfoASM{i}.xError;
300     timeASM(i)=solnInfoASM{i}.time;
301 end
302
303 %for five variables: omit 7, 9, and 13
304 % errorASM(13)=[];
305 % errorASM(9)=[];
306 % errorASM(7)=[];
307 % xErrorASM(13)=[];
308 % xErrorASM(9)=[];
309 % xErrorASM(7)=[];
310 % timeASM(13)=[];
```

```
311 % timeASM(9)=[];
312 % timeASM(7)=[];
313
314 errorStats=struct;
315 errorStats.meanVal=mean(errorASM);
316 errorStats.medianVal=median(errorASM);
317 errorStats.stdDevVal=std(errorASM);
318
319 timeStats=struct;
320 timeStats.mean=mean(timeASM);
321 timeStats.median=median(timeASM);
322 timeStats.stdDev=std(timeASM);
323
324 polyStats=struct;
325 polyStats.mean=mean(numPolys);
326 polyStats.median=median(numPolys);
327 polyStats.stdDev=std(numPolys);
```

Appendix D Chapter 4 Code

sec:BnBcode

D.1 The Multiparametric Subgradient Approximation Algorithm

The following is the code which was written for the mpSA; it is only used in D.2. It uses the mpASM as a subroutine, so needs to be in the same folder as that code.

```
1 function [xStar, uStar, vStar, fStar, CR, solnInfo]=mpSA(uStart,...
2     vStart,tol,Theta,problem,maxIter,varargin)
3 %This function assumes that the constraints are polynomial in the parameter
4 %-------------------------------------------------------------------------
5 % input                type                description
6 % uStart:              cell                initial dual variables
7 % vStart:              cell                initial dual eq variables
8 % tol:                 scalar              error tolerance
9 % Theta:               polyhedron          parameter space
```

```
10 % problem:                struct                problem to be solved
11 % varargin:               cell                  optional cap for CR splits
12 %
13 % output
14 % xStar:                  cell                  optimal decision function
15 % uStar:                  cell                  optimal dual function
16 % vStar:                  cell                  optimal dual function
17 % fStar:                  cell                  optimal value function
18 % CR:                     polyhedron vector     parameter space partition
19 % solnInfo:               struct                solution information
20 %-------------------------------------------------------------------------
21
22 %problem contains linear constraints (inequality and equality), constraints
23 %to be relaxed (inequality and equality), and lower and upper bounds
24 %linear constraints:
25 %A                        matrix                coefficient matrix
26 %b                        vector                constants
27 %E                        matrix                rhs parameter coefficient
28 %                                                  matrix
29
30 %relaxed constraints:
31 %relConIneq                cell                  quadratic anonymous
32 %                                                  inequality functions
33 %relConEq                  cell                  quadratic anonymous
34 %                                                  equality functions
35 %nonLinCon                function              nonlinear constraints for
36 %                                                  error calculations
37
38 %solnInfo fields:
39 %nlps:                    double                total nlps solved
40 %feasNLPs:                double                nlps for making u feasible
41 %dualIter:                double                number of dual updates
42 %tol:                     double                tolerance level
```

```
43 %error:                    double                    total error
44
45 if isempty(varargin)
46     asmCap=5;
47 else
48     asmCap=varargin{1};
49 end %if
50
51 %add in maxCR later as a varargin, but right now, just make it 1000
52 if 1
53     maxCR=1000;
54 else
55     maxCR=varargin{2};
56 end
57
58 %initialize
59
60 solnInfoK.nlps=0;
61 solnInfoK.feasNLPs=0;
62 solnInfoK.dualIter=0;
63 solnInfoK.numTooSmallPolys=0;
64
65 nConIneq=length(problem.relConIneq);
66 nConEq=length(problem.relConEq);
67 nCon=nConIneq+nConEq;
68
69 %critical regions for x
70 CRK=Theta.copy;
71 %critical regions for u and v; these should be the same
72 if nConIneq
73     CRuK=Theta.copy;
74 end %if
75
```

```
76  if nConEq
77      CRvK=Theta.copy;
78  end %if
79
80  iter=1;
81  %lambdaK=1/(iter+1);
82  lambdaK=1/(3*(iter+1));
83
84  % totalVol=Theta.volume();     %doesn't work if Theta is multiple CRs
85  totalVol=sum(Theta.volume());
86  normXiK=inf;
87
88  %initialize the cell of dual variables; vK might be empty
89  uK=uStart(:);
90  vK=vStart(:);
91
92  %need to be same size as CR
93  nCR=length(CRK);
94  xK=cell(nCR,1);
95  phiK=cell(nCR,1);
96
97  %save the constraints function until the end
98  errorCon=problem.constraints;
99
100 %mpASM only accepts lb and ub as columns; therefore modify the bounds
101 problem.lb=reshape(problem.lb,[],1);
102 problem.ub=reshape(problem.ub,[],1);
103
104 %run until tolerance is met
105 while normXiK> tol*totalVol && iter<maxIter+1 && length(CRK)<maxCR
106     disp(['iteration ' num2str(iter) ' of ' num2str(maxIter)]);
107     %xK is a cell corresponding to CRK, each cell contains a matrix
108     %containing rows [a,b] representing a*theta+b, where row i is x_i
```

```
109
110     %for each piece of the polyhedron, compute the updated solution
111     CRTemp=CRK.copy;
112     if nConIneq
113         uKTemp=uK;
114     end
115     if nConEq
116         vKTemp=vK;
117     end
118     for i=1:length(CRTemp)
119         if nConEq && nConIneq
120             problemK=relaxProb([uKTemp{i},vKTemp{i}], nConIneq, problem);
121         elseif nConEq
122             problemK=relaxProb(vKTemp{i}, nConIneq, problem);
123         else
124             problemK=relaxProb(uKTemp{i}, nConIneq, problem);
125         end %if
126 %% compute mpASM for the relaxed primal
127         %note that if polyhedra get too small, split doesn't work anymore
128         %and just returns an empty polyhedron object; this causes problems
129         %fixed as of midJune 2020
130         [x,phi,CR,solnInfo]=mpASM(tol,CRTemp(i),problemK,asmCap);
131
132         xK=[xK;x];
133         phiK=[phiK;phi];
134         %need to simplify
135         CRK=[CRK;CR];
136
137         solnInfoK.nlps=solnInfoK.nlps+solnInfo.nlps;
138         solnInfoK.numTooSmallPolys=solnInfoK.numTooSmallPolys+...
139                                         solnInfo.numTooSmallPolys;
140         %delete the old elements;
141         xK(1)=[];
```

```
142             phiK(1)=[];
143             CRK(1)=[];
144
145             %append the corresponding dual functions;
146             if nConIneq
147                 uK=[uK;repmat(uKTemp(i),size(CR))];
148             end
149
150             if nConEq
151                 vK=[vK;repmat(vKTemp(i),size(CR))];
152             end
153
154             %delete the old elements of the dual variables and
155             %update the dual critical regions
156             if nConIneq
157                 uK(1)=[];
158                 CRuK=CRK.copy;
159             end
160             if nConEq
161                 vK(1)=[];
162                 CRvK=CRK.copy;
163             end
164         end %end for
165
166     %% get the subgradient
167         nCR=length(CRK);
168         xiK=cell(nCR,1);
169
170         %to maintain consistency with Leverenz' code, xiK needs to be a
171         %linear interpolation of all the values at the vertices
172
173         for i=1:nCR
174             %get the vertices of the critical region: [vertices' ; 1']
```

```
175         vertices=CRK(i).V;
176         M=[vertices'; ones(1,length(vertices))];
177         nParam=CRK(i).Dim+1;
178         %get the xK values at the extreme points
179         xVal=xK{i}*M;
180         Minv=M^-1;
181         for j=1:nConIneq
182             %evaluate the relaxed inequality constraints at the vertices
183             gVal=zeros(1,nParam);
184             for k=1:nParam
185                 %gets the kth row from vertices and the kth column from
186                 %xVal
187                 gVal(k)=problem.relConIneq{j}(xVal(:,k),...
188                     vertices(k,:));
189             end %for
190             %for the ith CR, find u_j(\theta)
191             xiK{i}(j,:)=gVal*Minv;
192         end %for j
193         if nConEq
194             for j=1:nConEq
195                 %evaluate the relaxed equality constraints at the vertices
196                 hVal=zeros(1,nParam);
197                 for k=1:nParam
198                     %gets the kth row from vertices and the kth column from
199                     %xVal
200                     hVal(k)=problem.relConEq{j}(xVal(:,k),...
201                         vertices(k,:));
202                 end %for
203                 %for the ith CR find v_j(\theta)
204                 xiK{i}(nConIneq+j,:)=hVal*Minv;
205             end %for j
206         end %if
207     end %for i
```

```
208     %xiK{i} is set up [grad(consIneq); grad(consEq)] of length
209     %nConIneq+nConEq
210
211     %the L2-norm of xiK over CRK, denoted ||xiK||
212     normXiK=approxNorm(xiK,CRK);
213
214     %the unprojected dual (potentially infeasible)
215     uHatK=cell(nCR,1);
216     if nConEq
217         vHatK=cell(nCR,1);
218     end %if
219
220     %the update direction
221     dK=cell(nCR,1);
222
223     %WARNING: this part might lead to problems
224     %need to update uK and CRuK to correspond to CRK
225     %third argument is finer (has a longer array) than second argument
226
227     %10-5-2020: even though both start out corresponding to CRK, when
228     %CRK changes at the mpASM computation, we need to know what CRs uK and
229     %vK used to be attached to; hence the necessity of CRuK and CRvK;
230     %think about instead passing index lists rather than whole polyhedra
231     if nConIneq && length(CRK)~=length(CRuK)
232         [uK,CRuK] = updatePartition(uK,CRuK,CRK);
233     end %if
234     if nConEq && length(CRK)~=length(CRvK)
235         [vK,CRvK] = updatePartition(vK,CRvK,CRK);
236     end %if
237     if nConEq && nConIneq && length(CRvK)~=length(CRuK)
238         %think about this more when we get the code working quicker
239         % if an error occurs due to mismatching, then include another
240         % if-block which checks to see is bigger, CRvK or CRuK, and then
```

```
241         % matches the smaller one to that
242         [vK,CRvK] = updatePartition(vK,CRvK,CRuK);
243         [uK,CRuK] = updatePartition(uK,CRuK,CRvK);
244     end %if
245     if nConIneq
246         if length(CRK)~=length(CRuK)
247             [xiK,~] = updatePartition(xiK,CRK,CRuK);
248         end
249     else
250         if length(CRK)~=length(CRvK)
251             [xiK,~] = updatePartition(xiK,CRK,CRvK);
252         end
253     end %if
254
255     %update uHatK and vHatK
256     for i=1:nCR
257         %this check makes sure that the dual variable doesn't get too
258         %'spiky'
259         if normXiK>=1
260             dK{i}=xiK{i}/normXiK;
261         else
262             dK{i}=xiK{i};
263         end
264         if nConIneq
265             uHatK{i}=uK{i}+lambdaK*dK{i}(1:nConIneq,:);
266         end %if
267         if nConEq
268             vHatK{i}=vK{i}+lambdaK*dK{i}(nConIneq+1:nCon,:);
269         end %if
270     end %for
271
272
273     %compute feasible version of uHatK and vHatK
```

```
274     if nConIneq
275         uKtemp=cell(0);
276         CRuKtemp=cell(0);
277     end
278
279     %initialize the primal parts
280     xKTemp=xK;
281     phiKTemp=phiK;
282     CRKTemp=CRK.copy;
283
284     %count the number of CRs moved through in the new primal set (includes
285     %the updated CRs, so can't just be i)
286     count=0;
287     if nConIneq
288         for i=1:nCR
289             count=count+1;
290             vertices=CRuK(i).V;
291             dualVertVals=zeros(length(vertices),nConIneq);
292             for j=1:length(vertices)
293                 dualVertVals(j,:)=uHatK{i}*[vertices(j,:)';1];
294             end
295             %check the dual at the vertices; if they are all positive, then
296             %they don't need to be made feasible; 10^-6 was arbitrary
297
298             dualPos=all(dualVertVals>=-10^-6);
299             if dualPos
300                 uKtemp=[uKtemp;uHatK{i}];
301                 CRuKtemp=[CRuKtemp;CRuK(i)];
302             %if they are all negative, then they need to not update
303             else
304                 [tempDual,tempCR]=makeFeasible(uHatK{i},CRuK(i),tol);
305
306                 %append tempDual and tempCR to the new cell
```

```
307                     uKtemp=[uKtemp;tempDual];
308                     CRuKtemp=[CRuKtemp;tempCR];
309
310                     numAdded=length(tempCR);
311                     %update xK, phiK, and CRK as well
312                     if numAdded>1
313                         %note that if i+1>end, then Matlab appends an empty
314                         %vector
315                         xKTemp=[xKTemp(1:count-1);repmat(xK(i),size(tempCR));...
316                             xK(i+1:end)];
317                         phiKTemp=[phiKTemp(1:count-1);repmat(phiK(i),size(tempCR))
                                ;...
318                             phiK(i+1:end)];
319                         CRKTemp=[CRKTemp(1:count-1);tempCR;CRK(i+1:end)];
320                         %a change has been made, so count has to be updated
321                         %permanently; the -1 offsets that we are changing one
322                         %CR to multiple
323                         count=count+numAdded-1;
324                     end
325                 end
326             end
327             %note v is a free variable so nothing has to be done
328         end %end if
329         xK=xKTemp;
330         phiK=phiKTemp;
331         CRK=CRKTemp.copy;
332
333         if nConIneq
334             CRuK=CRuKtemp.copy;
335             uK=uKtemp;
336         end %if
337         if nConEq
338             vK=vHatK;
```

```
339     end %if
340     if nConEq && nConIneq && length(CRvK)~=length(CRuK)
341         % if an error occurs due to mismatching, then include another
342         % if-block which checks to see is bigger, CRvK or CRuK, and then
343         % matches the smaller one to that
344         [vK,CRvK] = updatePartition(vK,CRvK,CRuK);
345         [uK,CRuK] = updatePartition(uK,CRuK,CRvK);
346     end %if
347
348
349     %update CRK to match CRuK (or CRvK if there are no inequalities)
350     CRphiK=CRK.copy;
351     if nConIneq && length(CRK)~=length(CRuK)
352         [xK,CRK] = updatePartition(xK,CRK,CRuK);
353     elseif nConEq && length(CRK)~=length(CRvK)
354         [xK,CRK] = updatePartition(xK,CRK,CRvK);
355     end
356     %update phiK to match the partition
357     if length(CRphiK)~=length(CRK)
358         [phiK,~] = updatePartition(phiK,CRphiK,CRK);
359     end
360     nCR=length(CRK);
361     iter=iter+1;
362     lambdaK=1/(3*(iter+1));
363
364 end %end while
365
366
367 %% compute the error:
368 problemK.constraints=errorCon;
369 error1=zeros(1,nCR);
370 error2=zeros(1,nCR);
371 thetaErr=cell(1,nCR);
```

```
372 xErr=cell(1,nCR);
373 fErr=zeros(1,nCR);
374
375 for i=1:nCR
376     %note that computeError comes from Leverenz' 'mpSimplexApproximation.m'
377     %append xK and the center of the region for the original guess to where
378     %maximum error occurs
379     CRCenter=mean(CRK(i).V)';
380     x0=[xK{i}*[CRCenter;1];CRCenter];
381
382
383     %get the x and theta at which the largest error occurs on CRK(i)
384 %     [xErr{i}, thetaErr{i},error1(i)]=computeError(problemK,CRK(i),x0,phiK{i
    });
385 %     error2(i)=abs(problemK.objective(xErr{i}',thetaErr{i})-problemK.
    objective(...
386 %         xK{i}'*[thetaErr{i};1],thetaErr{i}));
387     [xErr{i}, thetaErr{i},error1(i)]=computeError(problem,CRK(i),x0,phiK{i});
388
389     %the xErr cells may need to be transposed, but if the objective
390     %requires them to be columns, then trouble erupts here;
391     %the trouble may be from non-joint-convexity;
392     %if the objective expects xErr to be a matrix (using x(i,:) for
393     %instance), then NOT transposing causes problems. Whee
394     fErr(i)=problem.objective(xErr{i},thetaErr{i});
395     error2(i)=abs(problem.objective(xErr{i},thetaErr{i})-...
396         problem.objective((xK{i}*[thetaErr{i};1]),thetaErr{i}));
397
398
399     %or, try to just use the midpoint of CRK to compute an error
400     %estimate
401 %     constr=@(x)problem.constraints(x,CRCenter);
402 %     centerObj=@(x) problem.objective(x,CRCenter);
```

```
403 %       bTemp=problem.b+problem.E*CRCenter;
404 %
405 %       %fmincon options
406 %       x0=xK{i}*[CRCenter;1];
407 %       options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
408 %       [xErr{i}, fMin(i)]=fmincon(centerObj,x0,problem.A,bTemp,[],[],...
409 %                                  problem.lb,problem.ub,constr,options);
410 %       xGurke=xK{i}*[CRCenter;1];
411 %       centerXErr{i}=abs(xGurke-xErr{i});
412 %       centerFErr(i)=problem.objective(xGurke,CRCenter)-fMin(i);
413 end %for
414
415
416 %% collect everything
417 %total the errors per critical region
418 %error2=sum(error2);
419 xStar=xK;
420 uStar=uK;
421 vStar=vK;
422 fStar=phiK;
423 CR=CRK.copy;
424 solnInfo=solnInfoK;
425 solnInfo.tol=tol;
426 solnInfo.dualIter=iter-1;
427 solnInfo.err1=error1;
428 solnInfo.err2=error2;
429 solnInfo.meanErr1=mean(error1);
430 solnInfo.meanErr2=mean(error2);
431 solnInfo.medianForErr=median(error1);
432 solnInfo.medianBackErr=median(error2);
433 solnInfo.thetaErr=thetaErr;
434 solnInfo.xErr=xErr;
435 solnInfo.fErr=fErr;
```

```
436
437 % solnInfo.fMin=fMin;
438 % solnInfo.centerXErr=centerXErr;
439 % solnInfo.centerFErr=centerFErr;
440 % solnInfo.gradNorm=normXiK;
441 end %end mpSA
442
443 %% extra functions
444 function [problemOut]=relaxProb(duals,nConIneq,problemIn)
445 %-------------------------------------------------------------------------
446 % input                 type                 description
447 % duals:                cell                 current dual variables
448 % nConIneq:             double               number of relaxed
449 %                                            inequalities
450 % problemIn:            struct               problem to be solved
451 %
452 % output
453 % problemOut:           struct               the relaxed problem
454 %-------------------------------------------------------------------------
455
456 %keep the affine inequalities, the bounds;
457 problemOut=problemIn;
458
459 %make the base constraints for mpASM to interact with
460 %WARNING: all code must now have a 'mpsaEx.baseConstraints=[];' line
461 problemOut.constraints=problemIn.baseConstraints;
462
463 %augment the objective function with the non-affine constraints
464 if length(duals)==nConIneq %there are no equality constraints
465     problemOut.objective= @(x,theta) lagrangeFunction(...
466         problemIn.objective,problemIn.relConIneq,duals,x,theta);
467 elseif nConIneq==0 %there are no inequality constraints
468     problemOut.objective= @(x,theta) lagrangeFunction(...
```

```
469             problemIn.objective,problemIn.relConEq,duals,x,theta);
470     else %there are both constraints
471         problemOut.objective= @(x,theta) lagrangeFunction(...
472             problemIn.objective,[problemIn.relConIneq,problemIn.relConEq],...
473             duals,x,theta);
474     end %if
475     end %relaxProb
476
477
478     %take the original objective function, and add slackness conditions to it
479     function outFunc=lagrangeFunction(objFunc,constraints,duals,x,theta)
480     %-----------------------------------------------------------------------
481     % input                 type                    description
482     % objFunc:              anon function           the objective function
483     % constraints:          cell                    the relaxing constraints
484     % duals:                matrix                  the dual variables
485     % x:                    col vector              the decision variables
486     % theta:                col vector              the parameter variables
487     %
488     % outputt
489     % outFunc:              scalar                  the Lagrange Function
490     %-----------------------------------------------------------------------
491
492     %note that
493     %u=[ aTheta1, b1 ; aTheta2, b2 ; ... ]
494     %u \in \R^{nCon\times kappa}
495
496     %make a cell of the different products below, element i being:
497     %vec{i}=u{i}*[theta; 1]*problem.relaxedConstraints{i}(x)*[theta; 1];
498     nCon=length(constraints);
499     vec=cell(nCon,1);
500     for i=1:nCon
501         %Note mpASM is transposing x, so we might be able to just transpose x,
```

```
502     %because, if x is a vector (column or row), then it won't matter, but
503     %if constraints treats x like a matrix (as it does when using
504     %matlabFunction), then the transpose is important
505     vec{i}=duals(i,:)*[theta;1]*constraints{i}(x,theta);
506 end %end for
507 outFunc=objFunc(x,theta)+sum([vec{:}]);
508 end %end lagrangeFunction
509
510 function [x]=funcProd(f,g,theta)
511 %-------------------------------------------------------------------------
512 % input                 type                  description
513 % f:                    anon function         the first input function
514 % g:                    anon function         the second input function
515 % theta                 col vector            the parameter variables
516 %
517 % output
518 % x:                                          f \dot g
519 %-------------------------------------------------------------------------
520     %assume that f and g are two matrices of equal size containing rows
521     %corresponding to functions affine in theta
522     [numRow,~]=size(f);
523     x=0;
524     for i=1:numRow
525         x=x+f(i,:)*[theta; 1]*g(i,:)*[theta; 1];
526     end %end for
527 end %end funcProd
528
529 function [dualFeas,CRFeas,solnInfo]=makeFeasible(dualHat,CR,tol)
530 %-------------------------------------------------------------------------
531 % input                 type                  description
532 % dualHat:              matrix                dual solution being checked
533 % CR:                   polyhedron            critical region
534 % tol:                  scalar                tolerance
```

```
535 %
536 % output
537 % dualFeas:              cell                feasible dual solution
538 % CRFeas:                polyhedron          feasible critical region
539 % solnInfo:              struct              info on solving the problem
540 %-------------------------------------------------------------------------
541 %
542 %     %this is for when uHat is a column vector
543 %     nCon=size(uHat,2);
544 %
545 %     %this is for when uHat is a row vector
546       nCon=size(dualHat,1);
547       %min ||u-uHat||^2 st u>=0
548       feasProb=struct;
549 %
550 %     note that u and uHat are row vectors (for the moment), but the columns
551 %     are separate variables
552 %     10-7-2020: use norm command here; it seems to get better results
553       feasProb.objective=@(dual,theta) (norm(dual-dualHat*[theta;1],2))^2;
554       feasProb.A=[];
555       feasProb.b=[];
556       feasProb.E=[];
557       feasProb.Aeq=[];
558       feasProb.beq=[];
559       feasProb.Eeq=[];
560       feasProb.constraints=[];
561       feasProb.lb=zeros(nCon,1);
562       feasProb.ub=inf*ones(nCon,1);
563
564       [dualFeas,~,CRFeas,solnInfo]=mpASM(tol,CR,feasProb);
565
566 end %end makeFeasible
567
```

```
568 function approxVol=approxNorm(xi,CR)
569 %copied and adapted from 'integrateQuadratic' by Jonathon Leverenz
570 %      CRtemp = CR.copy;
571 %
572 %      for i=1:length(CR)
573 %          CRtemp(i).addFunction(QuadFunction(f{i,1},f{i,2},f{i,3}),'f');
574 %      end
575 %      total = CRtemp.integrate('f');
576 %      total = sum(total);
577       %tempFunc=@(theta)funcProd(f,f,theta);
578
579       CRtemp=CR.copy;
580       numCR=length(CRtemp);
581
582       %note xi{i}(j,:)*[theta;1] is the linear
583       %approximation to relaxed constraint j at CR i;
584       %We need to dot xi against itself; since xi is a cell
585       %of linear coefficients with each cell corresponding to a CR,
586       %we have that, at CR(i), f{i}=xi{i}^T*xi{i}
587
588       %f{:,1} is the quadratic term,
589       %f{:,2} is the linear term,
590       %f{:,3} is the constant term
591       f=cell(numCR,3);
592       for i=1:numCR
593           [f{i,1}, f{i,2}, f{i,3}]=gurkeProd(xi{i});
594           CRtemp(i).addFunction(QuadFunction(f{i,1},f{i,2},f{i,3}),'func');
595       end
596       approxVol=CRtemp.integrate('func');
597       approxVol=sqrt(sum(approxVol));
598
599 end %end approxNorm
600
```

```
601 %a function which takes the matrix xi which represents a set of functions
602 % affine in theta and extracts the quadratic, linear, and constant
603 % coefficients from the inner product of it with itself
604 function [Q, A, c]=gurkeProd(xi)
605     [~, columns]= size(xi);
606     Q=zeros(columns-1);
607     A=zeros(1,columns-1);
608     for j=1:columns-1
609         for k=j:columns-1
610             Q(j,k)=xi(:,j)'*xi(:,k);
611             if j~=k
612                 Q(k,j)=Q(j,k);
613             end %if
614         end %for
615         A(j)=2*xi(:,j)'*xi(:,columns);
616     end %for
617     c=xi(:,columns)'*xi(:,columns);
618 end %gurkeProd
619
620 function [x1New, CRnew] = updatePartition(x1,C1o,C2o)
621 %Based on Leverenz' matchPartition function
622 %takes piecewise function x1 with partition described by C1o (for old) and
623 %updates it to fit C2o instead; this is only useful because C1o has been
624 %re-partitioned into C2o by the primal update step
625 %note that by the program above that C2o should always be a longer list
626 %than C1o
627 %-------------------------------------------------------------------------
628 %INPUT              TYPE              DESCRIPTION
629 %x1:                cell              function
630 %C1o:               polyhedron array  function space partition
631 %C2o:               polyhedron array  second space partition
632 %-------------------------------------------------------------------------
633 %OUTPUT             TYPE              DESCRIPTION
```

```
%x1New:                    cell                  function
%CRnew:                polyhedron array          new space partition
%-------------------------------------------------------------------------
n1 = length(C1o);
n2 = length(C2o);

%which part of x1 is x1New(index1(i)); since length(x1New)==length(C2),
%some of index1 will be duplicates
index1 = zeros(n2,1);

%switch; make i the outer loop so that at the end, if index(i)==0, set it
%to be the previous one, to avoid errors, since this is clearly a numerical
%issue
for j=1:n2
    for i=1:n1
        %if CR2(i) is in CR1(j), then append x1(j)
        if C2o(j)<=C1o(i)
            index1(j)=i;
        end
    end %for i
    if index1(j)==0
        index1(j)=index1(j-1);
    end
end %for j

x1New=x1(index1,:);
CRnew=C2o;
end %updatePartition

function f = nlpObjective(u,n,z,func)
%objective function for mp-nlp with parameters treated as variables
x = u(1:n);
p = u(n+1:end);
```

```
667 %these two calculations seem the same
668 %f = - abs(func(x,p)-z*[p;1]);
669 %f = z*[p;1]-func(x,p);
670
671 %transpose x (should work if x is actually considered a vector (column or
672 %row)) so that func still works while treating x as a matrix (like how
673 %matlabFunction does)
674 f = func(x,p)-z*[p;1];
675 end
676
677 function [c,ceq] = nlpConstraints(u,n,func)
678 %Nonlinear constraints for mp-nlp with parameters treated as variables
679 x = u(1:n);
680 p = u(n+1:end);
681 %transpose because if func expects a matrix, column vectors will break it
682 [c,ceq] = func(x,p);
683 end
```

D.2 The Comparison Example

sec:BnBcompare

The code for Section 4.3.4 is given below.

```
1  addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
       Matlab codes\mp ASM algorithm\mpASM')
2  addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
       MPT Practice\mpSAPractice')
3
4  %optimization problem structure
5  mpsaEx = struct;
6  %nonquadratic objective function handle
7  mpsaEx.objective = @(x,p)(400*(p(1)*x(1))^2+9*x(2)^2);
8  maxIter=20;
9
10 %linear constraint matrices (inequality and equality)
```

```
11 %Ax<=b+E\theta i.e. 2*x(1)+x(2) <= 2+theta
12 mpsaEx.A = [2,1];
13 mpsaEx.b = [2];
14 mpsaEx.E = [1];
15 mpsaEx.Aeq = [];
16 mpsaEx.Eeq = [];
17 mpsaEx.beq = [];
18
19 mpsaEx.baseConstraints=[];
20
21 %nonlinear constraints (for error computation, only)
22 mpsaEx.constraints=@nonlcon;
23
24 %constraints to be relaxed
25 mpsaEx.relConIneq{1}=@(x,theta) (x(1)-2)^2+(x(2)-2)^2-4-theta;
26 mpsaEx.relConEq=cell(0);
27 % mpsaEx.relConEq{1}=@(x,theta) (x(1)-2)^2+(x(2)-2)^2-4-theta;
28 % mpsaEx.relConIneq=cell(0);
29
30
31 %lower and upper bounds for x
32 mpsaEx.lb = zeros(2,1);
33 mpsaEx.ub = inf*ones(2,1);
34
35 Theta = Polyhedron('lb',[0.1],'ub',[1.1]);
36 %Theta = Polyhedron('lb',[0],'ub',[0.5]);
37
38 %triangulate Theta, take one simplex, and use an approximation for u0.
39 tempSimplex=Theta.triangulate.V;
40 vertices=tempSimplex;
41 %tempU=cell(length(vertices),1);
42
43 uStart=[];
```

```
44 %at each vertex, compute the solution and get the dual variable;
45 x0=mpsaEx.lb;
46 A=mpsaEx.A;
47 options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
48 for i=1:length(vertices)
49     b=mpsaEx.b+mpsaEx.E*vertices(i,:)';
50     obj=@(x) mpsaEx.objective(x,vertices(i,:));
51     const=@(x) mpsaEx.constraints(x,vertices(i,:));
52     [~,~,~,~,tempU,~,~]=fmincon(obj,x0,A,b,[],[],mpsaEx.lb,mpsaEx.ub,...
53         const,options);
54     uStart=[uStart;tempU.ineqnonlin'];
55 end %for
56
57 %compute the linear interpolation for u0
58 M=[vertices, ones(length(vertices),1)];
59 Minv=M^-1;
60 %transpose so that we have each row being the interpolation
61 uInterp= (Minv*uStart)';
62 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
63 u0{1}=uInterp;
64 %u0{1}=[0, 0];
65 v0={};
66
67 [x01,u01,v01,f01,CR01,solnInfo01]=mpSA(u0,v0,0.01,Theta,mpsaEx,maxIter);
68
69 %run the same example to see how straight mpASM compares
70 [x01ASM,f01ASM,CR01ASM,solnInfo01ASM] = mpASM(0.01,Theta,mpsaEx);
71
72 %make a grid pattern of the actual solution function
73 gridLength=31;
74 theta=linspace(0.1,1.1,gridLength);
75
76 %make cells of the solutions
```

```
77  xGrid=cell(2,1);
78  xGrid{1}=zeros(1,gridLength);
79  xGrid{2}=zeros(1,gridLength);
80
81  uGrid=cell(2,1);
82  uGrid{1}=zeros(1,gridLength);
83  uGrid{2}=zeros(1,gridLength);
84
85  fGrid=zeros(1,gridLength);
86  x0=mpsaEx.lb;
87
88  options = optimoptions('fmincon','Algorithm','sqp','Display','Off');
89  for i=1:length(theta)
90      A=mpsaEx.A;
91      b=mpsaEx.b+mpsaEx.E*theta(i);
92      obj=@(x) mpsaEx.objective(x,theta(i));
93      const=@(x) mpsaEx.constraints(x,theta(i));
94      [temp,fGrid(i),~,~,tempDual,~,~]=fmincon(obj,x0,A,b,[],[],mpsaEx.lb,mpsaEx
            .ub,...
95          const,options);
96      xGrid{1}(i)=temp(1);
97      xGrid{2}(i)=temp(2);
98      uGrid{1}(i)=tempDual.ineqnonlin(1);
99  %       uGrid{2}(i)=tempDual.ineqnonlin(2);
100 end %for
101
102 figure
103 plot(theta,fGrid,'o');
104 plotParametricFunction(f01,CR01);
105 title('Optimal Value Function mpSA');
106 xlabel('\theta');
107 ylabel('f(\theta)');
108
```

```
figure
plotParametricFunction(x01,CR01);
plot(theta,xGrid{1},'ko');
plot(theta,xGrid{2},'kx');
title('Optimal Solution Function mpSA');
xlabel('\theta');
ylabel('x(\theta)');

figure
plotParametricFunction(u01,CR01);
plot(theta,uGrid{1},'ko');
% plot(theta,uGrid{2},'kx');
title('Optimal Dual Function mpSA');
xlabel('\theta');
ylabel('u(\theta)');

figure
plot(theta,fGrid,'o');
plotParametricFunction(f01ASM,CR01ASM);
title('Optimal Value Function mpASM');
xlabel('\theta');
ylabel('f(\theta)');

figure
plotParametricFunction(x01ASM,CR01ASM);
plot(theta,xGrid{1},'ko');
plot(theta,xGrid{2},'kx');
title('Optimal Solution Function mpASM');
xlabel('\theta');
ylabel('x(\theta)');
```

```
%the nonlinear constraint for steepObj
function [g, h] = nonlcon(x,theta)
    g=(x(1)-2)^2+(x(2)-2)^2-4-theta;
```

```
    h=[];
end
```

D.3 The Multiparametric Branch-and-Bound Algorithm

Please note that the local for-loop index 'tufts' refers to the fluff at the edges of longhaired cats (such as their feet, ears, tail, etc.). Being a fringe element of the cat, it has been designated a semi-foul word by the author, and therefore fit for use when encountering numerical issues resulting in mis-identifying dimension in the Polyhedron class. Note, too, that there are a couple commented-out lines which plot figures; these were used to get the intermediate figures found in 4.4.2.

```
function [xstar,ystar,fstar,CR,solnInfo] = mpBnB(tol,Theta,problem,varargin)
%This function solves an mp-M01NLP using a branch-and-bound algorithm and
%applying an extension of the simplex approximation algorithm of
%Bemporad and Filippi (2006) at each node
%-------------------------------------------------------------------------
%INPUT                  TYPE                DESCRIPTION
%tol:                   scalar              error tolerance
%Theta:                 polyhedron          parameter space
%problem:               struct              mp-NLP information
%varargin:              cell: scalar        optional array of bounds for
%                                               integer constraints
%                       cell: scalar        optional cap for CR splits
%                       cell: chars         optional error type: 'BOM'
%                                               'LEM'
%-------------------------------------------------------------------------
%OUTPUT
%xstar:                 cell                optimal continuous decision
    function
%ystar:                 cell                optimal integer decision function
%fstar:                 cell                optimal value function
%CR:                polyhedron vector       partition of parameter space
%solnInfo:              struct              solution information
%-------------------------------------------------------------------------
```

```
23 %solnInfo Fields:
24 %error:      sum of largest single point errors for each simplex
25 %intError:  approximation of integrated error over parameter space
26 %nlps:       total number of NLP problems solved
27 %tol:        maximum error tolerance used
28 %numTooSmallPolys:       total number of Polyhedra that were not splittable
29 %time:       number of milliseconds to run
30 %-------------------------------------------------------------------------
31
32 %problem fields:
33 %---------------
34 % constraints=@(x,y,theta)
35 % objective=@(x,y,theta)
36 % lb
37 % ub
38 % Ax, Ay
39 % E
40 % b
41 % Aeqx, Aeqy
42 % Eeq
43 % beq
44 bnbTime=tic; %begin counting time
45 if isempty(varargin)
46     intBounds=1;
47     cap=5;
48     errorType='BOM';
49 elseif length(varargin)==1
50     intBounds=varargin{1};
51     cap=5;
52     errorType='BOM';
53 elseif length(varargin)==2
54     intBounds=varargin{1};
55     cap=varargin{2};
```

```
56      errorType='BOM';
57  else
58      intBounds=varargin{1};
59      cap=varargin{2};
60      errorType=varargin{3};
61  end %if
62
63  %for all examples, this was hardcoded in
64  cap=3;
65  testingListato=[];   %a temporary list containing all the nodes put in the
        graph
66
67  %the number of integer variables; this is stored in Ay, which should be an
68  %nCon-by-numInt matrix
69  ylen=size(problem.Ay,2);
70
71  %the number of parameters; this is stored in E, which should be an
72  %nCon-by-numParams matrix
73  numParams=size(problem.E,2);
74
75  %initialise the solution info
76  solnInfo.error=0;
77  solnInfo.nlps=0;
78  solnInfo.numTooSmallPolys=0;
79  solnInfo.tol=tol;
80  solnInfo.nodesList={};   %a list of the specific nodes solved
81  solnInfo.nodesSolved=0;
82  solnInfo.time=0;
83
84  %set up a structure for the solutions; it won't be filled until all the
85  %integer variables are set
86  solution=struct;
87  solution.intVars={Inf};
```

```
88  solution.ctsVars={Inf};
89  solution.value={Inf};
90  solution.CR=Theta;
91  solution.errorList=[Inf]; %list of the error terms associate with the CRs
92
93  %the max number of nodes at any time is going to be the depth, that is
94  %log2(length(y)), since calculations are going to be popped off as they're
95  %computed.
96  %the information we store in each element of integerNodeStack is {Inf, 0,
97  %1}; if Inf, then that means that that integer variable is relaxed
98  stackLen=ceil(log2(ylen));
99  integerNodeStack=zeros(stackLen,ylen);
100
101 %make a set of nodes; each node is a structure containing the same fields
102 %as the solution structure,
103 nodeStack=cell(1,stackLen);
104 tempNode=struct;
105 tempNode.integers=[]; %doesn't need to be a cell
106 tempNode.continuous={};
107 tempNode.value=Inf;
108 tempNode.CR=Theta;
109
110 %list of CRs at the node to be passed to the next node;
111 newNodeCRs=[];
112
113 relaxedProb=struct;
114
115 nodeStack{1}=tempNode;
116 tempIntMarker=Inf*ones(1,ylen);
117 integerNodeStack(1,:)=tempIntMarker;
118 stackPointer=2;
119
120 %calculate the answers at each node and make the comparisons;
```

```
121 %we don't ever need to calculate the solution at the all-continuous node,
122 %but we add it to the stack to start the process (via the stack being
123 %non-empty)
124 while ~isempty(integerNodeStack)
125
126     solveProb=0; %start out not needing to solve the problem
127
128     %pop one of the nodes off the stack, optimize the program at it,
129     %and compare
130     stackPointer=stackPointer-1;
131     tempNode=nodeStack{stackPointer};
132     nodeStack{stackPointer}=[];
133     %store tempNode.integers as a column to match the format of the
134     %continuous variables
135     tempNode.integers=integerNodeStack(stackPointer,:)';
136     integerNodeStack(stackPointer,:)=[];
137
138     %calculate the relaxed solution at workNode
139     %this really only needs to be done if there's actually a current
140     %integer solution stored with which to compare, that is, if any of
141     %the solution values are less than infinity--if not, all CRs of
142     %the node have to be marked as good
143
144     %workaround to apply 'any' to a cell
145     cellTester=@(x) any(x<Inf);
146     %if there are any parts of the solution that are finite then we solve
147     %the problem at this current node to compare against the solution
148
149     if any(cellfun(cellTester,solution.value))
150         solveProb=1;
151         isSoln=1;
152     else
153     %otherwise, there is no solution with which to compare
```

```
154         isSoln=0;
155         newNodeCRs=tempNode.CR;
156     end
157     %We also need to solve the problem if the current node has no relaxed
158     %integer variables; that is, we are at a leaf node
159     if all(tempNode.integers < Inf)
160         solveProb=1;
161         leafCompare=1;
162     else
163     %otherwise we are at a branch node;
164         %check if there is a solution; if there is no solution, we should
165         %NOT solve the problem
166         if isSoln
167             solveProb=1;
168         else
169             solveProb=0;
170         end
171         %regardless, this is not a leaf comparison
172         leafCompare=0;
173     end
174
175     %once a solution is actually found, compare; for the leafCompare
176     %comparison, we fiddle with the solution, while otherwise, if
177     %an intersection has a part where the node's solution is
178     %better than the solution, then it is added to the list of
179     %CRs which are solved at the next nodes;
180
181     if solveProb
182
183         solnInfo.nodesSolved=solnInfo.nodesSolved+1;
184         nodeX={};
185         nodeF={};
186         nodeCR=[];
```

```
187         nodeErrorList=[];
188
189         %clear out all nodes lacking enough vertices
190         killMe=zeros(size(tempNode.CR));
191         for tempNodeIdx=1:length(tempNode.CR)
192             killMe(tempNodeIdx)=size(tempNode.CR(tempNodeIdx).V,1) ...
193                 < numParams+1;
194         end
195         killMe=boolean(killMe);
196         tempNode.CR(killMe)=[];
197         %add the information from tempNode to problem so that the problem
198         %can be solved;
199         %this needs to be done in a loop over each CR in tempNode.CR
200         solnInfo.nodesList=[solnInfo.nodesList, tempNode.integers];
201         for tempNodeIdx=1:length(tempNode.CR)
202             %define relaxedProb
203             %relaxedInts is the location of all relaxed integers; we're
204             %taking advantage of the fact that we are relaxing the integers
205             %from y(1) to y(end), so tempIntMarker(~relaxedInts) is y(1),
206             %then y(1:2), then y(1:3), and so on
207             relaxedInts=tempNode.integers==Inf;
208             %2-13-23: relaxedInts needs to be a row vector, not a column
209             %vector
210             relaxedInts=relaxedInts';
211             constInts=tempNode.integers~=Inf;
212             numRelInts=sum(relaxedInts);
213             numCurrentInts=ylen-numRelInts;
214
215             %relax the nonlinear constraints and objective; x(1:n) is the
216             %original continuous variables, while x(n+1:end) is the relaxed
217             % integer variables, if there are any
218             n=size(problem.Ax,2);
219             %this is equivalent to checking leafCompare
```

```
220             if numCurrentInts<ylen
221             %The variables here need to be columns for the randomly
222             %generated instances; the x0 for mpASM is already a column, so
223             %we don't transpose x(1:n); we DO transpose the integers list
224             relaxedProb.constraints=@(x,theta)problem.constraints(x(1:n),...
225                 [tempNode.integers(constInts); x(n+1:end)],theta);
226             relaxedProb.objective=@(x,theta)problem.objective(x(1:n),...
227                 [tempNode.integers(constInts); x(n+1:end)],theta);
228             else
229             relaxedProb.constraints=@(x,theta)problem.constraints(x(1:n),...
230                 tempNode.integers(constInts),theta);
231             relaxedProb.objective=@(x,theta)problem.objective(x(1:n),...
232                 tempNode.integers(constInts),theta);
233             end
234             if isempty(problem.constraints)
235                 relaxedProb.constraints=[];
236             end
237
238             %add the relaxed bounds; this is one of the things that would
239             %have to change to extend this BnB algorithm to apply to a
240             %general mixed-integer program
241             relaxedProb.lb=[problem.lb; zeros(numRelInts,1)];
242             relaxedProb.ub=[problem.ub; ones(numRelInts,1)];
243
244             %relax the linear inequality constraints
245             relaxedProb.E=problem.E;
246             if ~isempty(problem.Ay)
247                 relaxedProb.A=[problem.Ax, problem.Ay(:,relaxedInts)];
248                 relaxedProb.b=problem.b - problem.Ay(:,~relaxedInts)*...
249                     tempNode.integers(constInts);
250             else
251                 relaxedProb.A=problem.Ax;
252                 relaxedProb.b=problem.b;
```

```
253             end
254
255             %relax the linear equality constraints
256             relaxedProb.Eeq=problem.Eeq;
257             if ~isempty(problem.Aeqy)
258                 relaxedProb.Aeq=[problem.Aeqx, problem.Aeqy(:,relaxedInts)];
259                 relaxedProb.beq=problem.beq-problem.Aeqy(:,~relaxedInts)*...
260                     tempNode.integers(constInts);
261             else
262                 relaxedProb.Aeq=problem.Aeqx;
263                 relaxedProb.beq=problem.beq;
264             end
265
266             [tempNodeX, tempNodeF, tempNodeSolnCR, tempNodeSI]=mpASM(tol,...
267                 tempNode.CR(tempNodeIdx),relaxedProb,cap,errorType);
268             nodeX=[nodeX;tempNodeX];
269             nodeF=[nodeF;tempNodeF];
270             nodeCR=[nodeCR; tempNodeSolnCR];
271             %we need the list of errors to be of corresponding length to
272             %nodeCR
273             nodeErrorList=[nodeErrorList,tempNodeSI.errorList];
274
275             solnInfo.nlps=solnInfo.nlps+tempNodeSI.nlps;
276             solnInfo.numTooSmallPolys=solnInfo.numTooSmallPolys+...
277                 tempNodeSI.numTooSmallPolys;
278             %7-25-22: think about how to add nodeSolnInfo error here
279             %7-27-22: tempNodeSolnInfo has a field errorList which
280             %corresponds to CR, storing the max error per simplex
281         end %tempNode.CR loop
282         nodeLen=length(nodeCR);
283         solnLen=length(solution.CR);
284
285 %this was used to pull some of the intermediate solutions for the
```

```
286 %illustrative examples
287 %           figure;
288 %           plotParametricFunction(nodeF,nodeCR);
289 %           xlabel('\theta_1');
290 %           ylabel('\theta_2');
291 %           zlabel('f(\theta_1,\theta_2)');
292 %           title(['integers: [' num2str(tempNode.integers')  ']']);
293
294           %if there is no solution, just replace the solution with the
295           %current node's solution; then continue the loop? not yet
296
297           nodeIdx=1;
298
299 %9-1-22: new plan: intersect first, record which parts intersected, and
300 %then compare over each of the intersections; this way there's no dynamic
301 %allocation of space.
302
303 %2-20-23: note that we need to also be able to keep
304 %the old solution which has NO intersection once we've fathomed part of the
305 %parameter space. We only need to do this at a leaf node because, at branch
306 %nodes, we haven't changed the solution yet. We do this with flags; if leaf
307 %node, save one flag for each CR in the solution; each flag is true if its
308 %corresponding CR has not been cut; that is, no intersection is
309 %full-dimensional
310           newSoln=struct;
311           newSoln.intVars={};
312           newSoln.ctsVars={};
313           newSoln.value={};
314           newSoln.CR={};
315           newSoln.errorList=[];  %list of the error terms at each CR
316
317 %reset the CRs at which to solve the children of a branch node to be empty
318 %and then add in all the regions where the node is better than the current
```

```
319 %solution
320         newNodeCRs=[];
321
322 %collect all the intersections between the solution and the node;
323         totalCaps=[];
324 %ith location of nodePtr contains the nodeIdx which resulted in the ith
325 %element of cap, similarly for solnPtr
326         nodePtr=[];
327         solnPtr=[];
328         %if comparing at a leaf, we need to keep the solutions which aren't
329         %intersected
330         if leafCompare
331             keepSoln=ones(1,solnLen);
332             leftoverSoln.CR=[];
333             leftoverSoln.intVars=[];
334             leftoverSoln.ctsVars=[];
335             leftoverSoln.value=[];
336             leftoverSoln.errorList=[];
337         end
338 %NOTE: one cannot append a truly empty [] to an 'empty polyhedron set'
339
340 %outer loop's gotta be solnIdx and inner's gotta be nodeIdx so that, after
341 %the solution at solnIdx is completely compared against all the nodes,
342 %what's left can be saved and added to newSoln
343         for solnIdx=1:solnLen
344             for nodeIdx=1:nodeLen
345                 captato=[];  %the intersection of the sets
346                 if(isFullDim(solution.CR(solnIdx)))
347                     captato=solution.CR(solnIdx).intersect(nodeCR(nodeIdx));
348                     captato(~isFullDim(captato))=[];
349                 end
350
351                 if(~isempty(captato))
```

```
352                     if leafCompare
353                         keepSoln(solnIdx)=false;
354                     end
355                     solnPtr=[solnPtr, repmat(solnIdx,1,length(captato))];
356                     nodePtr=[nodePtr, repmat(nodeIdx,1,length(captato))];
357                     totalCaps=[totalCaps; captato];
358                 end
359             end
360             if leafCompare
361                 if ~isempty(totalCaps)
362                     tempField.CR=solution.CR(solnIdx)\totalCaps;
363                     tempLength=length(tempField.CR);
364                     %do all the leftoverSoln fields as well, using repmat
365                     tempField.intVars=repmat(solution.intVars(solnIdx),
                            tempLength,1);
366                     tempField.ctsVars=repmat(solution.ctsVars(solnIdx),
                            tempLength,1);
367                     tempField.value=repmat(solution.value(solnIdx),tempLength
                            ,1);
368                     tempField.errorList=repmat(solution.errorList(solnIdx),1,
                            tempLength);
369
370                     for tempIdx=1:length(tempField.CR)
371                     if isFullDim(tempField.CR(tempIdx)) && size(tempField.CR(
                            tempIdx).V,1) >= numParams +1
372                         leftoverSoln.CR=[leftoverSoln.CR; tempField.CR(tempIdx
                                )];
373                         leftoverSoln.intVars=[leftoverSoln.intVars; tempField.
                                intVars(tempIdx)];
374                         leftoverSoln.ctsVars=[leftoverSoln.ctsVars; tempField.
                                ctsVars(tempIdx)];
375                         leftoverSoln.value=[leftoverSoln.value; tempField.
                                value(tempIdx)];
```

```
376                         leftoverSoln.errorList=[leftoverSoln.errorList,
                                tempField.errorList(tempIdx)];
377                     end
378                     end
379                 end
380             end
381         end
382
383         %compare on each intersection
384         capLen=length(totalCaps);
385         for capIdx=1:capLen
386             nodeIdx=nodePtr(capIdx);
387             solnIdx=solnPtr(capIdx);
388             potate=totalCaps(capIdx);
389
390             rhs=potate.b;
391             lhs=potate.A;
392             %cut up the constraints and take out the constant
393             %side f(node)-f(soln) <= 0 append to A\theta <= b
394             newConstraint=nodeF{nodeIdx}-solution.value...
395                 {solnIdx};
396
397             %fdiff(theta) + fdiff(const) <= 0 becomes newLHS <= newRHS
398             newRHS=-newConstraint(end);
399             newLHS=newConstraint(1:end-1);
400             if newRHS==Inf
401                 %the whole CR is better
402                 nodeBetterCR=Polyhedron(lhs,rhs);
403                 solnBetterCR=[];
404             else
405                 %the whole CR might still be better
406                 rhs=[rhs;newRHS];
407                 lhs=[lhs;newLHS];
```

```
408                 nodeBetterCR=Polyhedron(lhs,rhs);
409                 solnBetterCR=potate\nodeBetterCR;
410                 solnBetterCR.minVRep;
411             end
412             nodeBetterCR.minVRep;
413             %stolen from 'isBounded' to try to force boundedness condition,
414             %because, somehow, minVRep sets nodeBetterCR to be unbounded
415
416             %this is the case because while computing minVRep, computeVRep
417             %gets called, which then calls isBounded, which then fails to
418             %properly compute the feasibility of the half-space, labelling
419             %the polytope as unbounded. This can be avoided by computing
420             %the vertices and then GIVING the polytope those vertices....
421             if ~isBounded(nodeBetterCR)
422                 nodeBetterCR=Polyhedron(nodeBetterCR.V);
423                 nodeBetterCR.minVRep;
424             end
425             numCRsLeft=length(solnBetterCR);
426
427         %two parts: where the node value is better, and where the solution
428         %value is better;
429             %leaf node
430             if leafCompare
431                 %store where the solution is better and where the node is
432                 %better
433                 if ~isempty(solnBetterCR)
434                     %if MPT thinks the bounded solution is unbounded, make
435                     %a new polyhedron
436                     if ~isBounded(solnBetterCR)
437                         %use a for loop because size can't take the
438                         %dimensions of multiple matrices at once
439                         for tuft=length(solnBetterCR):-1:1
440                             if size(solnBetterCR(tuft).V,1) >= numParams +1
```

```
441                     solnBetterCR(tuft)=Polyhedron(solnBetterCR(tuft).V);
442                     solnBetterCR(tuft).minVRep;
443                         else
444                             solnBetterCR(tuft)=[];
445                         end
446                     end
447                 end
448             end
449             %need to remove all facets etc.
450             if isFullDim(nodeBetterCR)
451             newSoln.CR=[newSoln.CR; nodeBetterCR; solnBetterCR];
452             newSoln.intVars=[newSoln.intVars; tempNode.integers; ...
453                 repmat(solution.intVars(solnIdx),...
454                 numCRsLeft,1)];
455             newSoln.ctsVars=[newSoln.ctsVars; nodeX(nodeIdx); ...
456                 repmat(solution.ctsVars(solnIdx),...
457                 numCRsLeft,1)];
458             newSoln.value=[newSoln.value; nodeF(nodeIdx); ...
459                 repmat(solution.value(solnIdx),...
460                 numCRsLeft,1)];
461             newSoln.errorList=[newSoln.errorList, ...
462                 nodeErrorList(nodeIdx), ...
463                 repmat(solution.errorList(solnIdx), ...
464                 1,numCRsLeft)];
465             else  %the nodeBetter is actually a facet; ignore it
466             newSoln.CR=[newSoln.CR; solnBetterCR];
467             newSoln.intVars=[newSoln.intVars; ...
468                 repmat(solution.intVars(solnIdx),...
469                 numCRsLeft,1)];
470             newSoln.ctsVars=[newSoln.ctsVars; ...
471                 repmat(solution.ctsVars(solnIdx),...
472                 numCRsLeft,1)];
473             newSoln.value=[newSoln.value; ...
```

```
474                     repmat(solution.value(solnIdx),...
475                     numCRsLeft,1)];
476                 newSoln.errorList=[newSoln.errorList, ...
477                     repmat(solution.errorList(solnIdx), ...
478                     1,numCRsLeft)];
479                 end
480
481
482             %branch node
483             %store all full-dimensional better CRs as the next nodeSet
484             %to compute on
485             else
486                 breadFox=nodeBetterCR(isFullDim(nodeBetterCR));
487                 if ~isempty(breadFox)
488                 newNodeCRs=[newNodeCRs; breadFox];
489                 end
490             end
491         end
492         if leafCompare
493             %store all unintersected CRs in the solution, then the partially
494             %intersected CRs; this should happen after all the checking is
495             %done; do the checking and pick up the scraps
496
497             %this line is solely to prevent failures at one dimension
498             keepSoln=logical(keepSoln);
499             if any(keepSoln)
500             newSoln.CR=[newSoln.CR; solution.CR(keepSoln)];
501             newSoln.intVars=[newSoln.intVars; solution.intVars(keepSoln)];
502             newSoln.ctsVars=[newSoln.ctsVars; solution.ctsVars(keepSoln)];
503             newSoln.value=[newSoln.value; solution.value(keepSoln)];
504             newSoln.errorList=[newSoln.errorList, solution.errorList(keepSoln)
                    ];
505             end
```

```
506
507             if ~isempty(leftoverSoln)
508             newSoln.CR=[newSoln.CR; leftoverSoln.CR];
509             newSoln.intVars=[newSoln.intVars; leftoverSoln.intVars];
510             newSoln.ctsVars=[newSoln.ctsVars; leftoverSoln.ctsVars];
511             newSoln.value=[newSoln.value; leftoverSoln.value];
512             newSoln.errorList=[newSoln.errorList, leftoverSoln.errorList];
513             end
514
515             %don't trust the fullDim calculation; instead, count number of
516             %vertices....
517             %use a for loop because size can't take the
518             %dimensions of multiple matrices at once
519             for tuft=length(newSoln.CR):-1:1
520                 if size(newSoln.CR(tuft).V,1) >= numParams+1
521                     newSoln.CR(tuft)=Polyhedron(newSoln.CR(tuft).V);
522                     newSoln.CR(tuft).minVRep;
523                 else
524                     newSoln.CR(tuft)=[];
525                     newSoln.intVars(tuft)=[];
526                     newSoln.ctsVars(tuft)=[];
527                     newSoln.value(tuft)=[];
528                     newSoln.errorList(tuft)=[];
529                 end
530             end
531             solution=newSoln;
532
533         end
534
535     else %the node does not need solving
536         %define nodeCR to be tempNode.CR if the node is NOT being solved,
537         %so that it can be passed to future children;
538         nodeCR=tempNode.CR;
```

```
539     end %need to solve the problem
540
541     %if the relaxation is always bad, then don't add the children nodes
542     if ~isempty(newNodeCRs)
543         %push the children of the current node onto the stack; they need
544         %all the CRs that are NOT bad, so we need to store newNodeCRs
545
546         if ~leafCompare
547         %get the current node
548         tempIntMarker=tempNode.integers;
549         %add its children to the stack; that is, change the first instance
550         %of Inf to 0 and 1
551         i=find(tempIntMarker==Inf,1);
552         for j=0:intBounds
553             tempIntMarker(i)=j;
554             %push tempIntMarker onto the stack
555             integerNodeStack(stackPointer,:)=tempIntMarker;
556
557             testingListato=[testingListato;tempIntMarker];
558
559             %add the list of CRs to nodeStack(stackPointer)
560             %using THETA\(THETA\newNodeCRs), there are fewer CRs to
561             %solve over
562             temptato=Theta\(Theta\newNodeCRs);
563             nodeStack{stackPointer}.CR=temptato;
564             stackPointer=stackPointer+1;
565
566         end %number of children nodes
567         end
568
569     end %nodeBad test
570
571 end %while
```

```
572
573 %set the output information
574 xstar=solution.ctsVars;
575 ystar=solution.intVars;
576 fstar=solution.value;
577 for i=1:length(solution.CR)
578     solution.CR(i)=Polyhedron(solution.CR(i).V);
579 end
580 CR=solution.CR;
581
582 %finish counting the solution info
583 solnInfo.error=sum(solution.errorList);
584 solnInfo.time=toc(bnbTime);
585
586 end %function
```

D.4 Case Examples

The code for Section 4.4.2 is given below. Both examples operate exactly the same as those found in Section B.2.

D.4.1 One Parameter

```
1 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      Matlab codes\mp ASM algorithm\mpASM')
2 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
      Matlab codes\mp ASM algorithm')
3
4 %power generation program with convex, nonpolynomial objective, a quadratic
5 %constraint, and parameters in general locations
6 %optimization problem structure
7 mpbnbEx = struct;
8 %quadratic objective function handle
9 mpbnbEx.objective = @(x,y,p) p*x(1)^3+x(2)^3+x(3)^3-2*x(1)*y(1)...
```

```
10      -3*x(2)*y(2)-4*x(3)*y(2);
11
12  maxIter=10;
13
14  %linear constraint matrices (inequality and equality)
15  %Ax+Ay<=b+E\theta e.g. x(1)+x(2)+x(3) +y(1)+y(2)>= 0
16  mpbnbEx.Ax = [1, 1, 1];
17  mpbnbEx.Ay = [1, 1];
18  mpbnbEx.b = [3];
19  mpbnbEx.E = [0];
20
21  mpbnbEx.Aeqx = [];
22  mpbnbEx.Aeqy = [];
23  mpbnbEx.Eeq = [];
24  mpbnbEx.beq = [];
25
26  %nonlinear constraints (for error computation, only),
27  mpbnbEx.constraints=@(x,y,theta) nonlcon3x(x,y,theta);
28
29  %unrelaxed nonlinear constraints for mpSA only
30  mpbnbEx.baseConstraints=[];
31
32  %lower and upper bounds for x
33  mpbnbEx.lb = zeros(3,1);
34  mpbnbEx.ub = inf*ones(3,1);
35
36  theta1LB=1;
37  theta1UB=2;
38
39  Theta = Polyhedron('lb',[theta1LB],'ub',[theta1UB]);
40
41  %run the same example to see how straight mpASM compares
42  [x01ASM,y01ASM,f01ASM,CR01ASM,solnInfo01ASM] = mpBnB(0.010,Theta,mpbnbEx);
```

```
figure
plotParametricFunction(f01ASM,CR01ASM);
xlabel('\theta');
ylabel('f(\theta)');
title('Optimal Value Function mpASM');

figure
plotParametricFunction(x01ASM,CR01ASM);
xlabel('\theta');
ylabel('x(\theta)');
title('Optimal Solution Function mpASM');
```

```
function [g,h] = nonlcon3x(x,y,theta)
g(1)= theta-1.5-theta*x(2)^2-y(2);
h=[];
end
```

D.4.2 Two Parameters Power Generation

```
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm\mpASM')
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm')

%power generation program with convex, nonpolynomial objective, a quadratic
%constraint, and parameters in general locations
%optimization problem structure
mpbnbEx = struct;
%quadratic objective function handle
mpbnbEx.objective = @(x,y,p) p(1)*exp(-x(1)) + 10*x(2)^2 -3*x(1)*y(1) - ...
     2*(p(2)-2)*x(2)*y(2);

maxIter=10;
% maxIter=5;
```

```
14
15 %linear constraint matrices (inequality and equality)
16 %Ax+Ay<=b+E\theta i.e. x(1)+x(2) >= 1
17 mpbnbEx.Ax = [1, 1];
18 mpbnbEx.Ay = [0, 0];
19 mpbnbEx.b = [1];
20 mpbnbEx.E = [0, 0];
21
22 mpbnbEx.Aeqx = [];
23 mpbnbEx.Aeqy = [];
24 mpbnbEx.Eeq = [];
25 mpbnbEx.beq = [];
26
27 %nonlinear constraints (for error computation, only),
28 mpbnbEx.constraints=@(x,y,theta) genlcon2x(x,y,theta);
29
30 %unrelaxed nonlinear constraints for mpSA only
31 mpbnbEx.baseConstraints=[];
32
33 %lower and upper bounds for x
34 mpbnbEx.lb = zeros(2,1);
35 mpbnbEx.ub = inf*ones(2,1);
36
37 theta1LB=0.1;
38 theta1UB=1.2;
39 theta2LB=1;
40 theta2UB=3.5;
41
42 Theta = Polyhedron('lb',[theta1LB,theta2LB],'ub',[theta1UB,theta2UB]);
43
44 %run the same example to see how straight mpASM compares
45 [x01ASM,y01ASM,f01ASM,CR01ASM,solnInfo01ASM] = mpBnB(0.010,Theta,mpbnbEx);
46 figure
```

```
47 plotParametricFunction(f01ASM,CR01ASM);
48 xlabel('\theta_1');
49 ylabel('\theta_2');
50 zlabel('f(\theta_1,\theta_2)');
51 title('Optimal Value Function mpASM');
52
53 figure
54 plotParametricFunction(x01ASM,CR01ASM);
55 xlabel('\theta_1');
56 ylabel('\theta_2');
57 zlabel('x(\theta_1,\theta_2)');
58 title('Optimal Solution Function mpASM');
```

```
1 function [g,h] = genlcon2x(x,y,theta)
2 g(1)= theta(2)*x(1)^2 +x(1)*y(2) + x(2) -4;
3 g(2)= x(1) -(1+y(1))*x(2) -0.5*theta(1);
4 h=[];
5 end
```

D.5 Benchmark Functions

This random instance generator code uses a different version of *makeConstraints3Param.m*, one that includes integer variables, to construct its nonlinear constraint files. It functions the same as those in C.2, but adds in integer variables as well.

```
1 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm\mpASM')
2 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    Matlab codes\mp ASM algorithm')
3 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice')
4
5 addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    MPT Practice\mpSAPractice\randFuncs')
```

```
addpath('C:\Users\apangia\Documents\Grad School\Clemson\Research\OR Researh\
    paperBnB\testExs')   %for makeconstraints3params
%Random Example (plots)
%three variables
numX=3;
%integer variables
numY=2;
% numY=3;
% numY=4;

%MUST BE COLUMN FORM RIGHT NOW
xSize=[numX,1];       %col form;
ySize=[numY,1];

x=sym('x',xSize);
assume(x, 'real');
y=sym('y',ySize);
assume(y,'integer');

%one parameter
syms theta1;
assume(theta1>=0.1 & theta1<=1.1);

%ten examples, with 1 nonlinear constraint each
numExamples=16;  % 2 ints
% numExamples=13;  % 3 ints
% numExamples=14;  % 4 ints;

numNonlinConstraints=1;

%the parameter space
thetaLower=0.1;
thetaUpper=1.1;
```

```
38 Theta = Polyhedron('lb',thetaLower,'ub',thetaUpper);
39
40 %depth for mp-ASM
41 cap=5;
42
43 %set the precision of the symbols
44 digits(5);
45
46 %set the seed so that there is some semblance of replicability
47 rng default;
48 seedNum=8+numX;
49 rng(seedNum);
50
51 %set the guaranteed feasible point
52 bSelect=ones(numX+numY,1);
53
54 %Set the linear constraint Ax + E*theta<=b
55 numLinCon=2;
56 AxSize=[numLinCon,numX];
57 AySize=[numLinCon,numY];
58 Esize=[numLinCon,1];    %1 being for theta1
59 cSize=[1,numLinCon];
60 %matrices to help ensure feasibility for the linear constraints
61 thetaLowerRep=repmat(thetaLower,numLinCon,1);
62 thetaUpperRep=repmat(thetaUpper,numLinCon,1);
63
64 %the theta-picking matrix for the linear matrix
65 RaSize=[numLinCon,numLinCon];
66 RcSize=[numLinCon,numLinCon];
67
68 %the matrices for generating the objective function and the quadratic
69 %constraints
70 %Note Q=P'P
```

```
71  Psize=[2*(numX+numY),2*(numX+numY)];
72  RpSize=Psize;
73  qSize=[1,numX+numY];
74
75  cQuadSize=[1,1];
76  RcQuadSize=[1,1];
77
78  mpsaEx=cell(1,numExamples);
79  mpsaExTemp=struct;
80
81  %solution fields
82  xSA=cell(1,numExamples);
83  uSA=cell(1,numExamples);
84  vSA=cell(1,numExamples);
85  fSA=cell(1,numExamples);
86  CRSA=cell(1,numExamples);
87  solnInfoSA=cell(1,numExamples);
88
89  xASM=cell(1,numExamples);
90  yASM=cell(1,numExamples);
91  fASM=cell(1,numExamples);
92  CRASM=cell(1,numExamples);
93  solnInfoASM=cell(1,numExamples);
94
95  %the constraints
96  gurke=cell(numExamples,numNonlinConstraints);
97  constraintsList=cell(numExamples,numNonlinConstraints);
98
99  %the objectives
100 objList=cell(numExamples,1);
101
102 for i=1:numExamples
103     disp(['Creating Example ' num2str(i) ' of ' num2str(numExamples)]);
```

```
104     mpsaExTemp.Ax=floor(randMat(AxSize,0,1)*1000)/1000;
105     mpsaExTemp.Ay=floor(randMat(AySize,0,1)*1000)/1000;
106     %make a matrix of numbers ranging from 0 to 1, to determine whether
107     %theta_i is present in the linear constraints; less than 0.5
108     %means not present
109     thetaSelect=rand(Esize);
110     mpsaExTemp.E=floor(randMat(Esize)*1000)/1000;
111     mpsaExTemp.E(thetaSelect<0.5)=0;
112
113     thetaSelect=rand(Esize);
114     mpsaExTemp.E=floor(randMat(Esize)*1000)/1000;
115     mpsaExTemp.E(thetaSelect>=0.5)=1;
116
117     %we need that b gives feasibility for ALL theta, so it needs to
118     %incorporate E*theta in some way; use the lower bounds when E(i,j) is
119     %negative, and use the upper bounds when E(i,j) is positive;
120     %since we are assuming inequality, in all likelihood, using bSelect=1
121     %for the y values ought to generate more feasibility when y=0 than
122     %infeasibility
123     thetaAdjust=zeros(Esize);
124
125     thetaAdjust(mpsaExTemp.E<0)=thetaLowerRep(mpsaExTemp.E<0);
126     thetaAdjust(mpsaExTemp.E>=0)=thetaUpperRep(mpsaExTemp.E>=0);
127     mpsaExTemp.b=[mpsaExTemp.Ax, mpsaExTemp.Ay]*bSelect+ ...
128         dot(mpsaExTemp.E,thetaAdjust,2);
129
130     mpsaExTemp.Aeqx = [];
131     mpsaExTemp.Aeqy = [];
132     mpsaExTemp.Eeq = [];
133     mpsaExTemp.beq = [];
134
135     %unrelaxed nonlinear constraints
136     mpsaExTemp.baseConstraints=[];
```

```
137
138     %make the objective function: f(x;theta)=[x^T,y^T]Q(theta)[x;y]+
139     %q^T(theta)[x;y]
140
141     %Q=P'(theta)*P(theta)
142     %P(theta)=P1+P2*Rp*theta
143     P1=floor(randMat(Psize,0,1)*1000)/1000;
144     P2=sym(floor(randMat(Psize,0,1)*1000)/1000);
145
146     %put theta1 or no parameter into the quadratic form of the
147     %objective function
148     thetaSelect=rand(Psize);
149     noneHere=thetaSelect<0.5;
150     thetaHere=thetaSelect>=0.5;
151     P2(noneHere)=0;
152     P2(thetaHere)=P2(thetaHere).*theta1;
153     %vpa: variable-precision-arithmetic
154     P=vpa(P1+P2);
155     Q=expand(P'*P);
156
157     %q'(theta)*[x;y]
158     q1=floor(randMat(qSize,0,1)*1000)/1000;
159     q2=sym(floor(randMat(qSize,0,1)*1000)/1000);
160
161     %put theta1 or no parameter into the linear portion of the
162     %objective function
163     thetaSelect=rand(qSize);
164     noneHere=thetaSelect<0.5;
165     theta1Here=thetaSelect>=0.5;
166     q2(noneHere)=0;
167     q2(theta1Here)=q2(theta1Here).*theta1;
168     q=vpa(q1+q2);
169
```

```
170     obj=vpa([x;x;y;y]'*Q*[x;y;x;y]+q*[x; y]);     %if x is a col
171     objList{i}=vpa([x;x;y;y]'*Q*[x;y;x;y]+q*[x; y]);
172     obj=matlabFunction(obj,'Vars',{theta1,x,y});
173
174     %objective function; inner variables must be different than syms
175     mpsaExTemp.objective=@(varX,varY,varTheta) ...
176     obj(varTheta(1),varX,varY);
177
178     for j=1:numNonlinConstraints
179         %Q=P'(theta)*P(theta)
180         %P(theta)=P1+P2*Rp*theta
181         P1=floor(randMat(Psize,0,1)*1000)/1000;
182         P2=sym(floor(randMat(Psize,0,1)*1000)/1000);
183         %put theta1, theta2, or no parameter into the quadratic form of the
184         %constraint function
185         thetaSelect=rand(Psize);
186         noneHere=thetaSelect<0.5;
187         theta1Here=thetaSelect>=0.5;
188         P2(noneHere)=0;
189         P2(theta1Here)=P2(theta1Here).*theta1;
190         P=vpa(P1+P2);
191         Q=expand(P'*P);
192
193         %q'(theta)*x
194         q1=floor(randMat(qSize,0,1)*1000)/1000;
195         q2=sym(floor(randMat(qSize,0,1)*1000)/1000);
196         %put theta1, theta2, or no parameter into the linear portion of the
197         %constraint function
198         thetaSelect=rand(qSize);
199         noneHere=thetaSelect<0.5;
200         theta1Here=thetaSelect>=0.5;
201         q2(noneHere)=0;
202         q2(theta1Here)=q2(theta1Here).*theta1;
```

```
203         q=vpa(q1+q2);
204
205         %c(theta)
206         cQuad1=floor(randMat(cQuadSize,0,1)*1000)/1000;
207         cQuad2=sym(floor(randMat(cQuadSize,0,1)*1000)/1000);
208         %put theta1, theta2, or no parameter into the constant portion of
209         %the objective function
210         thetaSelect=rand(cQuadSize);
211         noneHere=thetaSelect<0.5;
212         theta1Here=thetaSelect>=0.5;
213         cQuad2(noneHere)=0;
214         cQuad2(theta1Here)=cQuad2(theta1Here).*theta1;
215         cQuad=vpa(cQuad1+cQuad2);
216
217         %b=bSelect'*Q(theta)*bSelect+q'(theta)*bSelect+cQuad(theta)
218         %make potate1=[x,x,y,y], potate2=[x,y,x,y]
219         potate1=[bSelect(1:numX);bSelect(1:numX);bSelect(numX+1:numX+numY);...
220             bSelect(numX+1:numX+numY)];
221         potate2=[bSelect(1:numX);bSelect(numX+1:numX+numY);bSelect(1:numX);...
222             bSelect(numX+1:numX+numY)];
223         b=potate1'*Q*potate2+q*bSelect+cQuad;
224
225         %generate the nonlinear constraints for mp-SA
226         gurke{i,j}=vpa([x;x;y;y]'*Q*[x;y;x;y]+q*[x; y]+cQuad-b);
227         constraintsList{i,j}=gurke{i,j};
228         gurke{i,j}=matlabFunction(gurke{i,j},'Vars',{theta1,x,y});
229         %constraints to be relaxed; inner variables must be different than
230         %syms
231         mpsaExTemp.relConIneq{j}=@(varX,varY,varTheta) ...
232             gurke{i,j}(varTheta(1),varX,varY);
233     end
234     %create the nonlcon constraint for mp-ASM
235     makeConstraints3Param(gurke(i),1,numX,numY,i);
```

```
236
237     mpsaExTemp.relConEq=cell(0);
238     %nonlinear constraints (for error computation, only)
239
240     %lower and upper bounds for x
241     mpsaExTemp.lb = zeros(xSize);
242     mpsaExTemp.ub = inf*ones(xSize);
243
244     mpsaEx{i}=mpsaExTemp;
245 end
246
247 %compute mp-BnB
248 for i=1:numExamples
249     %compute the example with the proper constraints
250     disp(['Solving Example ' num2str(i) ' of ' num2str(numExamples)]);
251     filename=['nonlcon' num2str(numX) 'Cts' num2str(numY) 'IntVarsEx'...
252         num2str(i) '.m'];
253     if isfile('nonlcon.m')
254         delete('nonlcon.m')
255     end
256     copyfile(filename, 'nonlcon.m');
257     mpsaEx{i}.constraints=@(x,y,theta) nonlcon(theta(1),x,y);
258
259     %run the same example to see how straight mpASM compares
260     [xASM{i},yASM{i},fASM{i},CRASM{i},solnInfoASM{i}] =...
261         mpBnB(0.010,Theta,mpsaEx{i});
262 end
263
264 for i=1:numExamples
265     disp(['Plotting Example ' num2str(i) ' of ' num2str(numExamples)]);
266     if(1)
267
268         gurkeFig=figure;
```

```
269             xlabel('\theta_1');
270             ylabel('f');
271             plotParametricFunction(fASM{i},CRASM{i});
272             title(['mpBnB-ASM ' num2str(numY) ' Int Vars Optimal Value Function'
                    ...
273                 ' for Example' num2str(i)]);
274             filename=['BnB_ASM' num2str(numY) 'IntVarsVal' num2str(i) '.fig'];
275             saveas(gurkeFig,filename);
276         end%if
277     end
278
279
280     %collect the data
281     errorASM=zeros(1,numExamples);
282     xErrorASM=zeros(1,numExamples);
283     timeASM=zeros(1,numExamples);
284     numPolys=zeros(1,numExamples);
285     nodesSolved=zeros(1,numExamples);
286
287     %we want the error to be average error, not total error: total error makes
288     %no sense given the number of polytopes that we could be encountering over
289     %higher dimensions
290     for i=1:numExamples
291         numPolys(i)=length(CRASM{i});
292         errorASM(i)=solnInfoASM{i}.error/numPolys(i);
293     %       xErrorASM(i)=solnInfoASM{i}.xError;
294         timeASM(i)=solnInfoASM{i}.time;
295         nodesSolved(i)=solnInfoASM{i}.nodesSolved;
296     end
297
298     %for two int variables:
299     %lose 1, 6, 9, 11, 13, 15
300     %keep 2,3,4,5,7,    8,10,12,14,16
```

```
301 errorASM([1,6,9,11,13,15])=[];
302 xErrorASM([1,6,9,11,13,15])=[];
303 timeASM([1,6,9,11,13,15])=[];
304 numPolys([1,6,9,11,13,15])=[];
305 nodesSolved([1,6,9,11,13,15])=[];
306
307 %for three int variables:
308 %lose 7,9,11
309 %keep 1:6,8,10,12,13
310 % errorASM([7,9,11])=[];
311 % xErrorASM([7,9,11])=[];
312 % timeASM([7,9,11])=[];
313 % numPolys([7,9,11])=[];
314 % nodesSolved([7,9,11])=[];
315
316 %for four int variables:
317 %lose 1,3,7,8
318 %keep 2,4,5,6,9,    10,11,12,13,14
319 % errorASM([1,3,7,8])=[];
320 % xErrorASM([1,3,7,8])=[];
321 % timeASM([1,3,7,8])=[];
322 % numPolys([1,3,7,8])=[];
323 % nodesSolved([1,3,7,8])=[];
324
325 errorStats=struct;
326 errorStats.meanVal=mean(errorASM);
327 errorStats.medianVal=median(errorASM);
328 errorStats.stdDevVal=std(errorASM);
329
330 xErrStats=struct;
331 xErrStats.mean=mean(xErrorASM);
332 xErrStats.median=median(xErrorASM);
333 xErrStats.stdDev=std(xErrorASM);
```

```
334
335 timeStats=struct;
336 timeStats.mean=mean(timeASM);
337 timeStats.median=median(timeASM);
338 timeStats.stdDev=std(timeASM);
339
340 polyStats=struct;
341 polyStats.mean=mean(numPolys);
342 polyStats.median=median(numPolys);
343 polyStats.stdDev=std(numPolys);
344
345 nodeStats=struct;
346 nodeStats.mean=mean(nodesSolved);
347 nodeStats.median=median(nodesSolved);
348 nodeStats.stdDev=std(nodesSolved);
```

```
1  function [] = makeConstraints3Param(constraints, numParams,...
2                                numCtsVars, numIntVars, nameIndex)
3  %create the constraints file that can then be read by fmincon
4  %-------------------------------------------------------------------------
5  %INPUT          TYPE                    DESCRIPTION
6  %constraints:   cell                    quadratic constraints
7  %numParams:     double                  number of parameters
8  %numVars:       double                  number of variables
9  %nameIndex:     double                  which example this is
10 %-------------------------------------------------------------------------
11 numCons=length(constraints);
12 charIdx=num2str(nameIndex);
13 fName=['nonlcon' num2str(numCtsVars) 'Cts' num2str(numIntVars)...
14     'IntVarsEx' charIdx '.m'];
15 fileID=fopen(fName,'w');
16
17 %note that matlabFunction calls a symbolic vector 'in#' where #represents
18 %the index at which it is located as soon as the symbolic vector is larger
```

```
%than 1x1. For instance, if theta1,theta2\in R and x\in R^2, then
%{theta1,theta2,x} becomes {theta1,theta2,in3} while {x,theta1,theta2}
%becomes {in1,theta1,theta2}
%technically, we should obtain where the vectors are located in the
%constraints, but I'm lazy: the code is forcing it to be 'in3' so that's
%what we'll make it
temp=['function [g,h] = nonlcon' num2str(numCtsVars) 'Cts'...
    num2str(numIntVars) 'VarsEx' charIdx];
if numParams==3
    firstLine=[temp '(theta1,theta2,theta3,in4,in5)\n'];
elseif numParams==2
    firstLine=[temp '(theta1,theta2,in3,in4)\n'];
else
    firstLine=[temp '(theta1,in2,in3)\n'];
end

fprintf(fileID, firstLine);
for i=1:numCons
    %find the first instance of ')' and then delete all parts of the array
    %prior to it; basically the '@('<argin>')'
    temp=char(constraints{i});
    closeParenIdx=strfind(temp,')');
    temp=temp(closeParenIdx(1)+1:end);
    constLine=['g(' num2str(i) ')=' temp ';\n'];
    fprintf(fileID, constLine);
end %for

%modify to accept equality constraints at some point
fprintf(fileID, 'h=[];\n');
fprintf(fileID, 'end');
fclose(fileID);

end %function
```

Bibliography

edo1996parametric [1] Joaquın Acevedo and Efstratios N Pistikopoulos. A parametric minlp algorithm for process synthesis problems under uncertainty. *Industrial & Engineering Chemistry Research*, 35(1):147–158, 1996.

97multiparametric [2] Joaquın Acevedo and Efstratios N Pistikopoulos. A multiparametric programming approach for linear process engineering problems under uncertainty. *Industrial & Engineering Chemistry Research*, 36(3):717–728, 1997.

vedo1999algorithm [3] Joaquın Acevedo and Efstratios N Pistikopoulos. An algorithm for multiparametric mixed-integer linear programming problems. *Operations Research Letters*, 24(3):139–148, 1999.

lgren2016solution [4] Nathan Adelgren. *Solution Techniques for Classes of Biobjective and Parametric Programs*. PhD thesis, Clemson University, 2016.

adelgren2020 [5] Nathan Adelgren. *Advancing Parametric Optimization: Theory and Solution Methodology for Multiparametric Linear Complementarity Problems with Parameters in General Locations*. SpringerBriefs on Optimization Series, 2021.

adjiman2000global [6] Claire S Adjiman, Ioannis P Androulakis, and Christodoulos A Floudas. Global optimization of mixed-integer nonlinear problems. *AIChE Journal*, 46(9):1769–1797, 2000.

lber1998nonsmooth [7] Ya.I. Alber, A.N. Iusem, and M.V. Solodov. On the projected subgradient method for nonsmooth convex optimization in a Hilbert space. *Mathematical Programming*, 81:23–35, 1998.

ulakis1995alphabb [8] Ioannis P Androulakis, Costas D Maranas, and Christodoulos A Floudas. αbb: A global optimization method for general constrained nonconvex problems. *Journal of Global Optimization*, 7(4):337–363, 1995.

tescu2009spectral [9] Mihai Anitescu. Spectral finite-element methods for parametric constrained optimization problems. *SIAM Journal on Numerical Analysis*, 47(3):1739–1759, 2009.

ascher2011first [10] Uri M. Ascher and Chen Greif. *A First Course on Numerical Methods*. SIAM, 2011.

ill2014parametric [11] Daniel Axehill, Thomas Besselmann, Davide Martino Raimondo, and Manfred Morari. A parametric branch and bound approach to suboptimal explicit hybrid MPC. *Automatica*, 50(1):240–246, 2014.

bank1982non [12] Bernd Bank, Jürgen Guddat, Diethard Klatte, Bernd Kummer, and Klaus Tammer. *Non-linear Parametric Optimization*. Springer, 1982.

Bednarczuk04 [13] E. Bednarczuk. Continuity of minimal points with applications to parametric multiple objective optimization. *European Journal of Operational Research*, 157:59–67, 2004.

orad2006algorithm [14] Alberto Bemporad and Carlo Filippi. An algorithm for approximate multiparametric convex programming. *Computational Optimization and Applications*, 35(1):87–108, 2006.

ben2009robust [15] Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. *Robust Optimization*. Princeton University Press, 2009.

benson85a [16] H.P. Benson. Multiple-objective linear programming with parametric criteria coefficients. *Management Science*, 31:461–474, 1985.

yamin2019applying [17] M.-Alizadeh Benyamin, Abdolali Basiri, and Sajjad Rahmany. Applying Gröbner basis method to multiparametric polynomial nonlinear programming. *Bulletin of the Iranian Mathematical Society*, 45(6):1585–1603, 2019.

ge1963topological [18] Claude Berge. *Topological Spaces*. Oliver and Boyd, 1963.

bitran80 [19] G.R. Bitran. Linear multi-objective programs with interval coefficients. *Management Science*, 26:694–706, 1980.

bonnel19 [20] H. Bonnel and C. Schneider. Post-Pareto analysis and a new algorithm for the optimal parameter tuning of the elastic net. *Journal of Optimization Theory and Applications*, 183:993–1027, 2019.

brenner2008fem [21] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Springer, 2008.

chankong1983 [22] V. Chankong and Y.Y. Haimes. *Multiobjective Decision Making: Theory and Methodology*. North-Holland Series in System Science and Engineering. North Holland, 1983.

os2020uncertainty [23] Vassilis M. Charitopoulos. *Uncertainty-Aware Integration of Control with Process Operations and Multi-Parametric Programming Under Global Uncertainty*. Springer Nature, 2020.

ostello2006random [24] Kevin P. Costello, Terence Tao, and Van Vu. Random symmetric matrices are almost surely nonsingular. *Duke Mathematical Journal*, 135(2):395–413, 2006.

ellnitz_witting09 [25] M. Dellnitz and K. Witting. Computation of robust Pareto points. *International Journal of Computing Science and Mathematics*, 2(3):243–266, 2009.

cvxpy [26] A. Diamond S., Agrawal and R. Murray. CVXPY. `https://www.cvxpy.org/examples/basic/quadratic_program.html`, 2020. Accessed March 3, 2021.

minguez2010recent [27] Luis F Domínguez, Diogo A Narciso, and Efstratios N Pistikopoulos. Recent advances in multiparametric nonlinear programming. *Computers & Chemical Engineering*, 34(5):707–716, 2010.

guez2013quadratic [28] Luis F. Domínguez and Efstratios N. Pistikopoulos. A quadratic approximation-based algorithm for the solution of multiparametric mixed-integer nonlinear programming problems. *AIChE Journal*, 59(2):483–495, 2013.

02multiparametric [29] Vivek Dua, Nikolaos A. Bozinis, and Efstratios N. Pistikopoulos. A multiparametric programming approach for mixed-integer quadratic engineering problems. *Computers & Chemical Engineering*, 26(4-5):715–733, 2002.

dua2004global [30] Vivek Dua, Katerina P Papalexandri, and Efstratios N Pistikopoulos. Global optimization issues in multiparametric continuous and mixed-integer optimization problems. *Journal of Global Optimization*, 30(1):59–89, 2004.

dua1999algorithms [31] Vivek Dua and Efstratios N. Pistikopoulos. Algorithms for the solution of multiparametric mixed-integer nonlinear optimization problems. *Industrial & Engineering Chemistry Research*, 38(10):3976–3987, 1999.

dua2000algorithm [32] Vivek Dua and Efstratios N Pistikopoulos. An algorithm for the solution of multiparametric mixed integer linear programming problems. *Annals of Operations Research*, 99(1-4):123–139, 2000.

dua2003parametric [33] Vivek Dua and Efstratios N. Pistikopoulos. Parametric optimization in process systems engineering: theory and algorithms. *Proceedings-Indian National Science Academy Part A*, 69(3/4):429–444, 2003.

Ehrgott2005 [34] M. Ehrgott. *Multicriteria Optimization*. Springer, 2005.

rgott2016multiple [35] M. Ehrgott, S. Greco, and J.R. Figueira. *Multiple Criteria Decision Analysis: State of the Art Surveys*. International Series in Operations Research and Management Science. Springer, 2nd edition, 2016.

ElBanna93 [36] A.Z.H. El-Banna. A study on parametric multiobjective programming problems without differentiability. *Comput. Math. Appl.*, 26(12):87–92, 1993.

chinchuluunetal08 [37] R. Enkhbat, J. Guddat, and A. Chinchuluun. Parametric multiobjective optimization. In A. Chinchuluun, P. Pardalos, A. Migdalas, and L. Pitsoulis, editors, *Pareto Optimality, Game Theory and Equilibria*, volume 17 of *Springer Optimization and Its Applications*, pages 529–538. Springer, 2008.

vans1970stability [38] James P Evans and Floyd J Gould. Stability in nonlinear programming. *Operations Research*, 18(1):107–118, 1970.

faisca2009global [39] Nuno P Faísca, Vassileios D Kosmidis, Berç Rustem, and Efstratios N Pistikopoulos. Global optimization of multi-parametric MILP problems. *Journal of Global Optimization*, 45(1):131–151, 2009.

Fang10 [40] Y.P. Fang and X.Q. Yang. Smooth representations of optimal solution sets of piecewise linear parametric multiobjective programs. In *Variational Analysis and Generalized Differentiation in Optimization and Control*, volume 47 of *Springer Optim. Appl.*, pages 163–176. Springer, New York, 2010.

co1976sensitivity [41] Anthony V. Fiacco. Sensitivity analysis for nonlinear programming using penalty methods. *Mathematical Programming*, 10(1):287–311, 1976.

o1983introduction [42] Anthony V. Fiacco. *Introduction to Sensitivity and Stability Analysis in Nonlinear Programming*. Elsevier, 1983.

co1990sensitivity [43] Anthony V. Fiacco and Yo Ishizuka. Sensitivity and stability analysis for nonlinear programming. *Annals of Operations Research*, 27(1):215–235, 1990.

acco1986convexity [44] Anthony V. Fiacco and Jerzy Kyparisis. Convexity and concavity properties of the optimal value function in parametric nonlinear programming. *Journal of Optimization Theory and Applications*, 48(1):95–126, 1986.

floudas1990global [45] Christodoulos A. Floudas and V. Visweswaran. A global optimization algorithm (GOP) for certain classes of nonconvex NLPs—I. Theory. *Computers & Chemical Engineering*, 14(12):1397–1417, 1990.

Galvan18 [46] E. Galvan, R.J. Malak, D.J. Hartl, and J.W. Baur. Performance assessment of a multiobjective parametric optimization algorithm with application to a multi-physical engineering system. *Structural and Multidisciplinary Optimization*, 58:489–509, 2018.

GEOFFRION1968 [47] A.M. Geoffrion. Proper efficiency and the theory of vector maximization. *Journal of Mathematical Analysis and Applications*, 22(3):618 – 630, 1968.

orski2007biconvex [48] Jochen Gorski, Frank Pfeuffer, and Kathrin Klamroth. Biconvex sets and optimization with biconvex functions: a survey and extensions. *Mathematical Methods of Operations Research*, 66(3):373–407, 2007.

guddat_etal85 [49] J. Guddat, F.G. Vasquez, K. Tammer, and K. Wendler. *Multiobjective and Stochastic Optimization Based on Parametric Optimization*, volume 26 of *Mathematical Research*. Akademie-Verlag, Berlin, 1985.

Haimes1971 [50] Y.Y. Haimes, L. Lasdon, and D Wismer. On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(3):296–297, 1971.

Galvan16 [51] D.J. Hartl, E. Galvan, R.J. Malak, and J.W. Baur. Parameterized design optimization of a magnetohydrodynamic liquid metal active cooling concept. *J. Mech. Des.*, 138:031402 (11 pages), 2016.

herceg2013multi [52] Martin Herceg, Michal Kvasnica, Colin N Jones, and Manfred Morari. Multi-parametric toolbox 3.0. In *2013 European Control Conference (ECC)*, pages 502–510. IEEE, 2013.

Hirschberger2010 [53] M. Hirschberger, Y. Qi, and R.E. Steuer. Large-scale MV efficient frontier computation via a procedure of parametric quadratic programming. *European Journal of Operational Research*, 204(3):581–588, 2010.

SteuerMarusQi13 [54] M. Hirschberger, R.E. Steuer, S. Utz, W. Wimmer, and Y. Qi. Computing the nondominated surface in tri-criterion portfolio selection. *Operational Research*, 61:169–183, 2013.

gan1973continuity [55] William W Hogan. The continuity of the perturbation function of a convex program. *Operations Research*, 21(1):351–352, 1973.

Huy08 [56] N.Q. Huy, B.S. Mordukhovich, and J.C. Yao. Coderivatives of frontier and solution maps in parametric multiobjective optimization. *Taiwanese J. Math.*, 12(8):2083–2111, 2008.

Jayasekara2019 [57] P.L.W. Jayasekara, N. Adelgren, and M.M. Wiecek. On convex multiobjective programs with application to portfolio optimization. *Journal of Multi-Criteria Decision Analysis*, 27(3-4):189–202, 2019.

ara2022parametric [58] Pubudu L. W. Jayasekara, Andrew Pangia, and Margaret M. Wiecek. On solving parametric multiobjective quadratic programs with parameters in general locations. *Annals of Operations Research*, 320:123–172, 2023.

johansen2002multi [59] T.A. Johansen. On multi-parametric nonlinear programming and explicit nonlinear model predictive control. In *Proceedings of the 41st IEEE Conference on Decision and Control*, volume 3, pages 2768–2773. IEEE, 2002.

leverenz2015 [60] J. Leverenz. *Network Target Coordination for Multiparametric Programming*. PhD dissertation, Clemson University, 2016.

Leverenz16 [61] J. Leverenz, M. Xu, and M.M. Wiecek. Multiparametric optimization for multidisciplinary engineering design. *Structural and Multidisciplinary Optimization*, 54(4):1–16, 2016.

verenz2015network [62] Jonathon Leverenz. *Network Target Coordination for Multiparametric Programming*. PhD thesis, Clemson University, 2015.

enz2022lagrangian [63] Jonathon Leverenz, Hyesuk Lee, and Margaret M. Wiecek. On Lagrangian duality for multiparametric programs. *Pacific Journal of Optimization*, 2022.

li2007MILP [64] Zukui Li and Marianthi G. Ierapetritou. A new methodology for the general multiparametric mixed-integer linear programming (MILP) problems. *Industrial and Engineering Chemistry Research*, 46(14):5141–5151, 2007.

Lucchetti00 [65] R.E. Lucchetti and E. Miglierina. Stability for convex vector optimization problems. *Optimization*, 53(5–6):517–528, 2004.

n1964inequalities [66] O. L. Mangasarian and J. B. Rosen. Inequalities for stochastic nonlinear programming problems. *Operations Research*, 12(1):143–154, 1964.

1976computability [67] Garth P McCormick. Computability of global solutions to factorable nonconvex programs: Part i—convex underestimating problems. *Mathematical Programming*, 10(1):147–175, 1976.

ay2005convergence [68] Khamron Mekchay and Ricardo H Nochetto. Convergence of adaptive finite element methods for general second order linear elliptic pdes. *SIAM Journal on Numerical Analysis*, 43(5):1803–1827, 2005.

meng2021objective [69] Zhiqing Meng, Min Jiang, Rui Shen, Leiyan Xu, and Chuangyin Dang. An objective penalty function method for biconvex programming. *Journal of Global Optimization*, pages 1–22, 2021.

mitsos2007milp [70] Alexander Mitsos and Paul I. Barton. Parametric mixed-integer 0—1 linear programming: The general case for a single parameter. *European Journal of Operational Research*, 194(3):663–686, 2008.

in2002convergence [71] Pedro Morin, Ricardo H. Nochetto, and Kunibert G. Siebert. Convergence of adaptive finite element methods. *SIAM Review*, 44(4):631–658, 2002.

Naccache79 [72] P.H. Naccache. Stability in multicriteria optimization. *J. Math. Anal. Appl.*, 68(2):441–453, 1979.

o2009developments [73] Diogo Alexandre Cipriano Narciso. *Developments in Non-Linear Multiparametric Programming and Control*. PhD thesis, Imperial College London, 2009.

Oberdieck16 [74] R. Oberdieck and N. Pistikopoulos. Multiobjective optimization with convex quadratic cost functions: A multiparametric programming approach. *Computers and Chemical Engineering*, 85:36–39, 2016.

erdieck2014branch [75] Richard Oberdieck, Martina Wittmann-Hohlbein, and Efstratios N Pistikopoulos. A branch and bound method for the solution of multiparametric mixed integer linear programming problems. *Journal of Global Optimization*, 59:527–543, 2014.

ohtake1985branch [76] Yoshiaki Ohtake and Naonori Nishida. A branch-and-bound algorithm for 0–1 parametric mixed integer programming. *Operations Research Letters*, 4(1):41–45, 1985.

angia2023biconvex [77] Andrew Pangia. Approximating optimal solutions to biconvex parametric programs. 2022. Revised and resubmitted, Optimization Letters.

dri1998parametric [78] Katerina P Papalexandri and Theodora I Dimkou. A parametric mixed-integer optimization algorithm for multiobjective engineering problems involving discrete decisions. *Industrial & Engineering Chemistry Research*, 37(5):1866–1882, 1998.

pappas2020exact [79] I. Pappas, N.A. Diangelakis, and E.N. Pistikopoulos. The exact solution of multiparametric quadratically constrained quadratic programming problems. *Journal of Global Optimization*, pages 1–27, 2020.

pappas2021exact [80] Iosif Pappas, Nikolaos A. Diangelakis, and Efstratios N. Pistikopoulos. The exact solution of multiparametric quadratically constrained quadratic programming problems. *Journal of Global Optimization*, 79(1):59–85, 2021.

penstern86 [81] J.P. Penot and A. Sterna-Karwat. Parametrized multicriteria optimization: Continuity and closedness of optimal multifunctions. *Journal of Mathematical Analysis and Applications*, 120(1):150–168, 1986.

penstern89 [82] J.P. Penot and A. Sterna-Karwat. Parametrized multicriteria optimization: Order continuity of the marginal multifunctions. *Journal of Mathematical Analysis and Applications*, 144(1):1–15, 1989.

dis1993parametric [83] Anastasios Pertsinidis. *On the Parametric Optimization of Mathematical Programs with Binary Variables and its Applications in the Chemical Engineering Process Synthesis.* PhD thesis, Carnegie Mellon University, 1993.

ikopoulos2002line [84] Efstratios N. Pistikopoulos, Vivek Dua, Nikolaos A. Bozinis, Alberto Bemporad, and Manfred Morari. On-line optimization via off-line parametric optimization tools. *Computers & Chemical Engineering*, 26(2):175–185, 2002.

Pistikopoulos21 [85] E.N. Pistikopoulos, N.A. Diangelakis, and R. Oberdieck. Multi-parametric Optimization and Control. Operations Research and Management Science. Wiley, 2021.

qiu2019explicit [86] Yiwei Qiu, Jin Lin, Feng Liu, and Yonghua Song. Explicit MPC based on the Galerkin method for AGC considering volatile generations. *IEEE Transactions on Power Systems*, 35(1):462–473, 2019.

rippa1992long [87] Shmuel Rippa. Long and thin triangles can be good for linear interpolation. *SIAM Journal on Numerical Analysis*, 29(1):257–270, 1992.

Romanko2012 [88] O. Romanko, A. Ghaffari-Hadigheh, and T. Terlaky. Multiobjective optimization via parametric optimization: Models, algorithms, and applications. In T. Terlaky and F.E. Curtis, editors, *Modeling and Optimization: Theory and Applications*, pages 77–119. Springer New York, 2012.

Ruetsch10 [89] G.R. Ruetsch. Using interval techniques to solve a parametric multi-objective optimization problem. *United States Patent No. 7.664,622 B2*, 2010.

Sawaragi85 [90] Y. Sawaragi, H. Nakayama, and T. Tanino. *Theory of Multiobjective Optimization.* Academic Press, 1985.

empfle2000optimal [91] Martin Stämpfle. Optimal estimates for the linear interpolation error on simplices. *Journal of Approximation Theory*, 103(1):78–90, 2000.

Steuer2011 [92] R.E. Steuer, Y. Qi, and M. Hirschberger. Comparative issues in large-scale mean-variance efficient frontier computation. *Decision Support Systems*, 51:250–255, 2011.

Thuan00 [93] L.V. Thuan and D.T. Luc. On sensitivity in linear multiobjective programming. *J. Optim. Theory Appl.*, 107(3):615–626, 2000.

waldron1998error [94] Shayne Waldron. The error in linear interpolation at the vertices of a simplex. *SIAM Journal on Numerical Analysis*, 35(3):1191–1200, 1998.

Wiecek16 [95] M.M. Wiecek and G. Dranichak. Robust multiobjective optimization for decision making under uncertainty and conflict. In J.C. Smith, editor, *Optimization Challenges in Complex, Networked, and Risky Systems*, Tutorials in Operations Research, pages 84–114. INFORMS, 2016.

wieceketal16 [96] M.M. Wiecek, M. Ehrgott, and A. Engau. Continuous multiobjective programming. In S. Greco, M. Ehrgott, and J. Figueira, editors, *Multiple Criteria Decision Analysis: State of the Art Surveys*, pages 738–815. Springer, 2nd edition, 2016.

Witting13 [97] K. Witting, S. Ober-Blöbaum, and M. Dellnitz. A variational approach to define robustness for parametric multiobjective optimization problems. *J. Global Optim.*, 57(2):331–345, 2013.

witting_etal08 [98] K. Witting, B. Schulz, M. Dellnitz, J. Böcker, and N. Fröhleke. A new approach for online multiobjective optimization of mechatronic systems. *International Journal of Software Tools and Technology Transfer*, 10:223–231, 2008.

ittmann2013global [99] M. Wittmann-Hohlbein and E.N. Pistikopoulos. On the global solution of multi-parametric mixed integer linear programming problems. *Journal of Global Optimization*, 57(1):51–73, 2013.

zewde2022novel [100] Addis Belete Zewde and Semu Mitiku Kassa. A novel approach for solving multi-parametric problems with nonlinear constraints. *Journal of Global Optimization*, 85:283–313, 2023.

ZhouHastie2005 [101] H. Zhou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society B*, 67(2):301–320, 2005.