5-2014

Indefinite Knapsack Separable Quadratic Programming: Methods and Applications

Jaehwan Jeong
University of Tennessee - Knoxville, jjeong3@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Jaehwan Jeong entitled "Indefinite Knapsack Separable Quadratic Programming: Methods and Applications." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Management Science.

Chanaka Edirisinghe, Major Professor

We have read this dissertation and recommend its acceptance:

Hamparsum Bozdogan, Bogdan Bichescu, James Ostrowski

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

5-2014

Indefinite Knapsack Separable Quadratic Programming: Methods and Applications

Jaehwan Jeong
University of Tennessee - Knoxville, jjeong3@utk.edu

Indefinite Knapsack Separable Quadratic Programming: Methods and Applications

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Jaehwan Jeong

May 2014

Dedicated to My Parents, Sister, Brother

and My lovely Wife

Acknowledgement

First and foremost, I would like to express my best gratitude to my advisor, Dr. Chanaka Edirisinghe. There is no doubt that he poured his heart into me. I am sure that I could not reach this professional level of the study without his sincere consideration and passion. He has the distinguished ability to train a student as a scientific thinker. When I first met him, I was like a wild horse that solely depends on its instinct and crude knowledge. He has patiently sharpened my thinking process and continuously showed his brilliant approaches toward research. Moreover, he has steadily demonstrated how a person can devote enthusiasm and professionalism for one's life. He also emphasizes cultivating not only excellent researchers but also "right" researchers who purse moral principles with ethical philosophy. I am really honored to have him as my advisor. He absolutely deserves to receive my full respect. If my future research adds any value for human beings, all the origin sources are from him.

I also want to give my special thanks to my dissertation committee members, Dr. Hamparsum Bozdogan, Dr. Bogdan Bichescu, and Dr. James Ostrowski. Their gentle hearts always cheered me up, and their luminous and explicit advice has guided me to increase the value of my research and life. My graduate study has always been enjoyable being with them, and their profound attitude for life and research are the role models to design my life.

I am grateful to my parents who initiated my dream for research. Their invaluable encouragements, endless supports, and unconditional love are the fundamental foundation of this dissertation. My sister and brother have also been constant sources of encouragement and wishes.

This dissertation could not have been accomplished without the dedication, support, and love of my beautiful wife Jungim Yoo. Her strong encouragement has been always the main source to keep me going forward. Her silent patience is definitely the substantial part of this dissertation. I love yoo (you)!

Lastly, the words in this acknowledgement would never be able to express my gratitude to people who supported me. This dissertation is just a piece of proof of their love.

Abstract

Quadratic programming (QP) has received significant consideration due to an extensive list of applications. Although polynomial time algorithms for the convex case have been developed, the solution of large scale QPs is challenging due to the computer memory and speed limitations. Moreover, if the QP is nonconvex or includes integer variables, the problem is NP-hard. Therefore, no known algorithm can solve such QPs efficiently. Alternatively, row-aggregation and diagonalization techniques have been developed to solve QP by a subproblem, knapsack separable QP (KSQP), which has a separable objective function and is constrained by a single knapsack linear constraint and box constraints.

KSQP can therefore be considered as a fundamental building-block to solve the general QP and is an important class of problems for research. For the convex KSQP, linear time algorithms are available. However, if some quadratic terms or even only one term is negative in KSQP, the problem is known to be NP-hard, i.e. it is notoriously difficult to solve.

The main objective of this dissertation is to develop efficient algorithms to solve general KSQP. Thus, the contributions of this dissertation are five-fold. First, this dissertation includes comprehensive literature review for convex and nonconvex KSQP by considering their computational efficiencies and theoretical complexities. Second, a new algorithm with quadratic time worst-case complexity is developed to globally solve the nonconvex KSQP, having open box constraints. Third, the latter global solver is utilized to develop a new bounding algorithm for general KSQP. Fourth, another new algorithm is developed to find a bound for general KSQP in linear time complexity. Fifth, a list of comprehensive applications for convex KSQP is introduced, and direct applications of indefinite KSQP are described and tested with our newly developed methods.

Experiments are conducted to compare the performance of the developed algorithms with that of local, global, and commercial solvers such as IBM Cplex using randomly generated problems in the context of certain applications. The experimental results show that our proposed methods are superior in speed as well as in the quality of solutions.

Table of Contents

1. Introduction **1**

1.1. Motivations for KSQP . . . 5
1.2. Research Scope . . . 6
1.3. Contributions of this Dissertation . . . 6
1.4. Outline of this Dissertation . . . 8
1.5. Notation and Preliminaries . . . 9

2. Knapsack Separable Quadratic Programming (KSQP): Survey **13**

2.1. Introduction . . . 13
2.2. Strictly Convex case . . . 16
2.3. Positive Semidefinite case . . . 53
2.4. Nonconvex case . . . 54
2.5. Conclusion . . . 61

3. Indefinite Open-Box Approach to Solution **62**

3.1. Introduction . . . 62
3.2. Open-Box Global (`OBG`) Solver . . . 63
3.3. Bounding with Open-Box Global (`OBG`) solver . . . 85
3.4. Experiments . . . 89
3.5. Conclusion . . . 97

4. Lagrangian-relaxed Closed-Box Solution **99**

4.1. Introduction . . . 99
4.2. Closed-Box Solver (`CBS`) . . . 100
4.3. Methodology and Implementation . . . 123
4.4. Experiments . . . 131
4.5. Conclusion . . . 162

5. Applications **165**

5.1. Applications for Convex case . . . 165
5.2. Applications for Nonconvex case . . . 167

6. Concluding Remarks **175**

Reference **179**

Appendix **195**

A. Appendix **196**

A.1. The case of $a_j = 0$ . . . 196

A.2. Transformation . . . 196
A.3. Transformation computation . . . 197
A.4. Another derivation of $x_j(\lambda)$ for $j \in N$ . . . 199
A.5. Compared solvers . . . 199
A.6. Initial expected λ by a sigmoid function . . . 202
A.7. Supplemental experiment results . . . 204

B. Implementation in Matlab **210**
B.1. Implementation in MATLAB . . . 210
B.2. Efficient coding in MATLAB . . . 212

Vita **216**

List of Tables

2.1. History of Algorithms for strictly convex case . . . 15
2.2. Safeguards of Newton method . . . 32
2.3. Minimum saved operations in Newton method . . . 36
2.4. The worst case number of operations in iteration parts . . . 42
2.5. History of experiment for strictly convex case . . . 49

3.1. Average CPU times for small problems using GN1$(n, \delta, 0, 0.5)$ . . . 93
3.2. Average CPU times for medium problems using GN1$(n, \delta, 0, 0.5)$ . . . 94
3.3. Average CPU times for large problems using GN1$(n, \delta, 0, 0.5)$ . . . 94
3.4. Sensitivity on ζ (average CPU times (sec.) for GN1$(n, 10, \zeta, 0.5)$) . . . 95
3.5. Sensitivity on ν (average CPU times (sec.) for GN1$(n, 10, 0, \nu)$) . . . 95
3.6. Relative speed of OBG on (n, δ, ζ, ν) . . . 96
3.7. Quality and speed of upper bounds on closed-box problems with GN2$(n, 10, 0, 0.5)$ 97

4.1. Hybrid methods for strictly convex case of (P) . . . 127
4.2. Range of breakpoints for test problems . . . 140
4.3. Instances by size of K^* and % of negative d_j . . . 142
4.4. Average seconds and average iteration by size for strictly convex case . . . 144
4.5. Average seconds by problem . . . 145
4.6. Average number of breakpoints after bracketing phase by problem . . . 146
4.7. Total number of safeguard used for each 500 instances . . . 146
4.8. Average seconds and average iteration by size for indefinite case . . . 149
4.9. Average number of breakpoints after bracketing phase . . . 149
4.10. Average seconds by problem . . . 150
4.11. Average time growth rate by % of negative d_j . . . 152
4.12. Average percentage in use of safeguard . . . 152
4.13. Total count of worse bound . . . 153
4.14. Top rankings . . . 153
4.15. Average seconds and range for small size indefinite problems . . . 156
4.16. Percentage of the best solution found for small size indefinite problems . . . 157
4.17. Average seconds and range for large size indefinite problems . . . 159
4.18. Percentage of the best solution found for large size indefinite problems . . . 160
4.19. Performance of CBS for GN2$(n, 10, 0, 0.5)$ . . . 162

5.1. Average CPU time (in seconds) for real investment portfolio selection . . . 171

A.1. Computational comparison of transformed and original problems . . . 198
A.2. Sensitivity of OBG on (n, δ, ζ, ν) . . . 204
A.3. Range of seconds by size and problem for strictly convex case . . . 205
A.4. Range of iteration by size and problem for strictly convex case . . . 206

A.5. Range of seconds by size and problem for indefinite case 207
A.6. Range of iteration by size and problem for indefinite case 208
A.7. Range of the number of breakpoints after bracketing phase 208
A.8. Average seconds by % of negative d_j 209

List of Figures

1.1. Motivation and research path 6

2.1. λ_l, λ_u, g_l, g_u, and tolerances 19
2.2. Secant method 25
2.3. Newton method 33
2.4. Lagrange multiplier search method 44
2.5. Separable and Non-separable case 45
2.6. Pegging method 46
2.7. Interval test method 47

3.1. $x_j(\lambda)$ for $j \in N$ 68
3.2. Global cases 72
3.3. Complexity curves on the proportion of $|N|$ 83
3.4. Behavior of lower and upper bounds by iteration 89
3.5. Average speed-up factor 95

4.1. $x_j(\lambda)$ is a non-increasing piecewise linear function on λ 104
4.2. $R(\lambda) = \sum_{\forall j} R_j(\lambda)$ is a continuous piecewise concave function 106
4.3. x_j for $j \in Z$ and $j \in N$ 107
4.4. $g(\lambda)$ is the right derivative (slope) of $R(\lambda)$ 111
4.5. $g(\lambda)$ 113
4.6. Suggested hybrid methods for Closed-box solver (`CBS`) 126
4.7. $g(\lambda)$ and folded sigmoid function 137
4.8. Histograms of breakpoints 139
4.9. Average speed by size for strictly convex case 143
4.10. Average speed by problem 145
4.11. Average speed by size for indefinite case 148
4.12. Average speed by problem 150
4.13. Average speed by % of negative d_j 151
4.14. Performance profile for small size indefinite problems 158
4.15. Performance profile for large size indefinite problems 161
4.16. λ^*_L and λ^*_U 164

5.1. Efficient frontier line 170
5.2. Cases that $Rel.error > 0.001$ 170
5.3. Relative quality of upper bounds (`RQUB`) on subset-sum problems 173
5.4. Average speed (second) in subset-sum problem 174

6.1. Global optimality depending on RHS b 177

B.1. Computation time for $s = \sum_{j \in I} a_j b_j$ 214

List of Algorithms

2.1. $O(n\ log\ n)$; Bisection method . . . 21
2.2. $O(n\ log\ n)$; Sorting method . . . 22
2.3. $O(n)$; Exact Median search method (Binary Search) . . . 23
2.4. Expected $O(n)$; Approximate median search method (Random search) . . . 24
2.5. Bracketing phase: Find a bracket $[\lambda_l, \lambda_u]$ . . . 26
2.6. Secant method: Do secant update . . . 27
2.7. $O(n^2)$; Pegging method (Variable fixing) . . . 28
2.8. $O(n^2)$; Newton method . . . 34
2.9. $O(n\ log\ n)$; Interval test method . . . 40
2.10. $O(n\ log\ n)$; Simplified version of `CKP*` of Moré and Vavasis [83] . . . 57
2.11. $O(n)$ time improved version of Moré and Vavasis [83] . . . 58

3.1. $O(|N|(n - |N|))$; Open-Box Global (`OBG`) solver . . . 76
3.2. Update procedure in *point* and *interval test* . . . 80
3.3. $O(1)$; `Update`: a sub function to update the best values . . . 81
3.4. $O(|J_Z^*|)$; Get x_j^* for $j \in J_Z^*$ distributing w . . . 82
3.5. Upper Bounding algorithm with `OBG` . . . 88

4.1. $O(n)$; Find a global optimum solution $\mathbf{x}^{U*}$ for the case of $K^* \subseteq Z$ . . . 119
4.2. Try to find the best $\mathbf{x}^{U*}$ in case of $K^* \cap Z \neq \emptyset$ and $K^* \cap N \neq \emptyset$, . . . 120
4.3. $O(n)$; A modified version of Algorithm 2.11 for $(P3)$ . . . 121
4.4. Try to find the best $\mathbf{x}^{U*}$ in the case of $K^* \subseteq N$ . . . 122
4.5. $O(n)$; Fixing algorithm . . . 126
4.6. Root finding type algorithm for indefinite case of (P) . . . 130
4.7. Breakpoint search type algorithm for indefinite case of (P) . . . 131
4.8. Interval test algorithm for indefinite case of (P) . . . 132

5.1. Mixed Integer programming with `CBS` (`MICBS`) . . . 169

1. Introduction

The quadratic programming (QP) is a mathematical optimization model with a quadratic objective function and linear constraints:

$$\begin{aligned} \text{(QP)} \quad & Min \quad \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ & s.t. \quad \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \qquad\; \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where $\mathbf{x} \in \mathbb{R}^n$ denote decision variables, and $\mathbf{c}$, $\mathbf{l}$, $\mathbf{u} \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m\times n}$, and $\mathbf{Q} \in \mathbb{R}^{n\times n}$ are data. In the box constraint, namely $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$, some u_j can be infinite $(+\infty)$ and some l_j can be infinite $(-\infty)$ as well. If all eigenvalues of $\mathbf{Q}$ are nonnegative (i.e. $\mathbf{Q}$ is positive definite or positive semidefinite), QP is a convex problem, and several polynomial time algorithms are available [24]. If $\mathbf{Q}$ has some negative eigenvalues (i.e. $\mathbf{Q}$ is indefinite or negative definite) or even one negative eigenvalue, QP is a very difficult problem (*NP-hard*) [104]. Convex QP has been extensively studied due to its myriad applications such as resource allocation, network flows, transportation, traffic scheduling, economies of scale, and portfolio selection. Moreover, a nonlinear optimization problem having a twice continuously differentiable objective function and constraints can be solved by the iterative use of convex QP by the method of successive quadratic programming. Indefinite QP is also appears in many applications such as VLSI (Very-large-scale integration) chip design and linear complementarity problems. See surveys [97, 110] and books [34, 42] for detailed applications and studies.

An example of convex QP is the index-tracking optimization model that finds the optimum investment weights to construct equity index funds by mimicking a sample of historical market return such as Standard & Poor's 500 index. For stocks $j = 1, 2...n$ and time periods $t = 1, 2, ..., T$, the model can be formulated in a least-square framework as

$$\begin{aligned} Min \quad & \sum_{t=1}^{T}\left[\sum_{j=1}^{n} R_{tj}x_j - r_t\right]^2 = (\mathbf{Rx} - \mathbf{r})^2 \\ s.t. \quad & \mathbf{1}^T\mathbf{x} = 1 \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the portfolio weight for stock j, R_{tj} is return of stock j at time t, r_t is the index (market) return at time t, $\mathbf{1} \in \mathbb{R}^n$ is a vector of 1, and $\mathbf{l},\ \mathbf{u} \in \mathbb{R}^n$ are given bound for the portfolio weights. See Edirisinghe [38] for more information and Nakagawa et al. [87] for models that consider regulatory.

Knapsack Nonseparable QP (KNQP)

Although much research has been conducted for QP, development of better QP solution algorithms is an ongoing challenge, because as the size of the problem increases the existing algorithms suffer from limitation in computer memory and computational speed. So, for specially structured QP with sparse or block angular matrix, efficient methods have been proposed. However, most methods still have the same difficulties as the problem size grows.

An attractive way to overcome the above computational difficulties is by row-aggregating the given set of linear constraints into a single linear constraint. Such an idea is the building-block of surrogation techniques [48, 87] and dual ascent methods [73, 134, 137]. A QP with a single linear constraint is called a knapsack QP. Moreover, when the matrix $\mathbf{Q}$ is non-diagonal, the QP is said to be nonseparable. Thus, we name the problem knapsack nonseparable QP (KNQP).

Because surrogation techniques and dual ascent methods solve KNQP iteratively, efficient algorithms for KNQP are required. Moreover, KNQP is very useful itself for various applications. For example, the convex KNQP are found in resource allocation, support vector machine (SVM) [31, 25], and portfolio selection [96]. The case of nonconvex KNQP can be found in applications involving maximum clique problem [105] and subset-sum problem [59, 132].

To solve convex KNQP, Pang (1980, [96]) employed the linear complementarity technique, and Dussault et al. (1986, [36]) and Lin and Pang (1987, [73]) proposed diagonalization technique (DT). Recently, spectral projected gradient (SPG) method of [12] was applied by Dai and Fletcher (2006, [31]). Research that specially focuses on indefinite KNQP is found in Pardalos et al. (1981, [105]), which applied interior point approach and simplicial partitioning approach.

Knapsack Separable QP (KSQP)

When $\mathbf{Q}$ is diagonal, QP is said to be separable, and the class of knapsack QP is referred to as knapsack separable QP (KSQP). For both convex and nonconvex cases, all available algorithms are discussed in the survey Chapter 2, and applications are discussed in Section 5.1.

When all diagonal elements of diagonal matrix $\mathbf{Q}$ are positive in KSQP, geometrically, the contours of the objective function are ellipsoids, and the optimum solution is the point at which the smallest ellipsoid is tangential to the feasible domain. Thus, strictly convex KSQP is a restricted projection problem [111]. For example, if $\mathbf{Q}$ is an identity matrix, the optimum solution is the closest (projection) point from the center of the ellipsoid (in this case, a sphere) to the feasible domain.

Due to this geometric nature and the existence of fast algorithms for convex KSQP (see Table 2.1 for algorithm references), it is used as a sub-problem in many QP algorithms. For example, DT and SPG approach to the solution iteratively projecting a point via KSQP onto its feasible domain. Therefore, KSQP is the fundamental building-block to solve KNQP with DT and SPG, and eventually, it can be used to develop algorithms for QP with row-aggregation techniques.

Indefinite QP

As mentioned earlier, indefinite QP is known to be *NP-hard*. It can be solved by some general global optimization techniques such as Benders decomposition, branch and bound, and cutting plane [99, 100], but there is a well known separable programming method that is specialized for QP (e.g. [114]). It first diagonalizes $\mathbf{Q}$ with a spectral decomposition,

$$\mathbf{Q} = \mathbf{V}\mathbf{D}\mathbf{V}^T,$$

where $\mathbf{D}$ is a diagonal matrix with eigenvalues of $\mathbf{Q}$ in its diagonal part, and $\mathbf{V}$ is the corresponding eigenvector matrix. With an appropriate constant vector $\mathbf{s}$, a linear mapping of

$$\mathbf{V}\mathbf{y} + \mathbf{s} \leftarrow \mathbf{x}$$

can transform indefinite QP to a form of separable QP:

$$\begin{aligned} Min \quad & \tfrac{1}{2}\mathbf{y}^T\mathbf{D}\mathbf{y} + \hat{\mathbf{c}}\mathbf{y} \\ s.t. \quad & \hat{\mathbf{A}}\mathbf{y} = \hat{\mathbf{b}} \\ & \boldsymbol{\alpha} \leq \mathbf{y} \leq \boldsymbol{\beta} \end{aligned} \tag{1.1}$$

where constants $\mathbf{s}$, $\boldsymbol{\alpha}$, $\boldsymbol{\beta} \in \mathcal{R}^n$ are determined by solving a linear program referred to multiple-cost-row linear program (see [114] for more details).

Since some diagonal elements of $\mathbf{D}$ are negative, (1.1) is still *NP-hard*. But the separable structure makes it possible to apply the piecewise linear approximation technique that formulates a mixed integer binary program; however, this formulation naturally increases the number of 0-1 variables exponentially as the number of variables and the level of accuracy increase. Thus, the technique is not practical for large size problems. See Section 2.4.1 for details of this formulation given in Pardalos and Kovoor [101].

1.1. Motivations for KSQP

Indefinite QP If the row-aggregation techniques are applied to indefinite separable QP in (1.1), it becomes indefinite KSQP. Thus, as an alternative to increasing the number of 0-1 variables exponentially by piecewise linear approximation, indefinite KSQP emerges as a sub-problem of a solution approach. In this sense, indefinite KSQP becomes an important class of problems for research.

Nonseparable QP As discussed earlier, the convex QP can be solved by KNQP with row-aggregation techniques, and KNQP algorithms such as DT and SPG utilize KSQP for their main sub-problem. In this respect, KSQP is a fundamental building-block to solve nonseparable QP so the efficient algorithm development for KSQP in this dissertation is beneficial to develop more efficient convex QP algorithms.

Mixed integer programming According to Pardalos [100], any nonlinear 0-1 program can be reduced to a quadratic program, and quadratic 0-1 programming can be formulated as a continuous global concave minimization problem. Because these converted problems can be solved via indefinite KSQP as a sub-problem with row-aggregation and diagonalization techniques, the value of research for KSQP is enlarged for mixed integer programming.

Indefinite KSQP Indefinite KSQP arises in various combinatorial problem formulations [35, 53, 60], in resource allocation [54], and in subset-sum problems [83, 132]. In spite of these valuable applications, indefinite KSQP is still a challenging problem as an *NP-hard* problem [59, 132]. Moreover, there exists only a handful of algorithms that focus on indefinite KSQP. Algorithms of [83, 132] finds a local optimum in polynomial time based on Karush-Kuhn-Tucker (KKT) conditions, and algorithms of [101] and [131] approximate a solution based on a mixed integer linear formulation and dynamic programming, respectively. Therefore, an efficient indefinite KSQP method is indispensable for the above applications, especially, when the problem size is large.

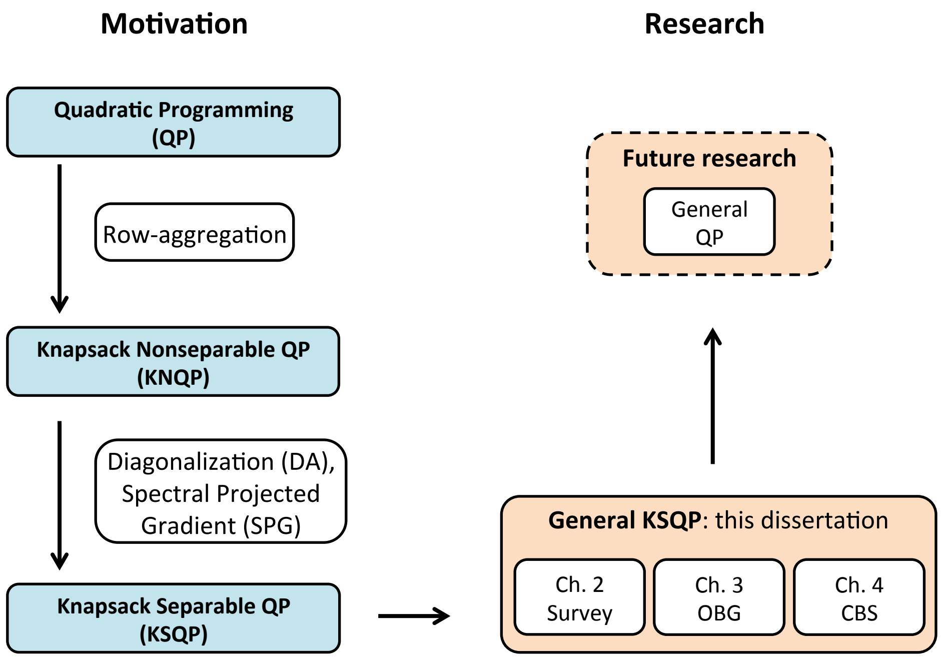

Motivation
Research
Quadratic Programming (QP)
Row-aggregation
Knapsack Nonseparable QP (KNQP)
Diagonalization (DA), Spectral Projected Gradient (SPG)
Knapsack Separable QP (KSQP)
Future research
General QP
General KSQP: this dissertation
Ch. 2 Survey
Ch. 3 OBG
Ch. 4 CBS

Figure 1.1.: Motivation and research path

1.2. Research Scope

As discussed earlier, KSQP can be used as a sub-problem in solving general QP. However, this dissertation is limited to the specific goal:

"Developing efficient algorithms to solve general KSQP."

The integration and the application of KSQP methods toward solving the general QP are outside the scope of this dissertation. See the flow chart in Figure 1.1.

1.3. Contributions of this Dissertation

1. **Survey of KSQP (Chapter 2)**

   A comprehensive literature review of all convex and nonconvex cases of KSQP is presented with theoretical and computational analysis. New ideas to improve the existing methods are also suggested.

2. **A new $O(n^2)$ time global algorithm for open-box indefinite KSQP; `OBG` (Section 3.2)**

   An algorithm, referred to Open-Box Global (`OBG`), is developed to find an exact global optimum solution of indefinite KSQP with open box constraints ($l_j \leq x_j$ for concave terms). Although `OBG` enumerates a subset of KKT points to determine a global optimum, the enumeration is made efficient by developing the theory for shrinking and partitioning the search domain of KKT multipliers, resulting $O(n^2)$ time complexity. This is the first instance of such an algorithm to the best of the author's knowledge. The superiority of `OBG` is verified computationally by comparing with other solvers such as the commercial solver Cplex 12.6.

3. **A new bounding algorithm for closed-box indefinite KSQP (Section 3.3)**

   The `OBG` method in Section 3.2 is utilized to generate lower and upper bounds of indefinite KSQP in the general case of closed box constraints ($l_j \leq x_j \leq u_j$ for concave terms). An efficient method utilizing a search on the Lagrange multiplier space is developed to find a lower bound, and a new procedure to construct a feasible upper bound is developed. It is computationally tested on random problems as well as subset-sum problems. It turns out that these bounds are quite tight and computationally very efficient for very large problems compared to Cplex 12.6.

4. **A new linear time bounding algorithm for closed-box indefinite KSQP; `CBS` (Chapter 4)**

   A new bounding algorithm, referred to Closed-Box Solver (`CBS`), is developed to find a tight bound of indefinite KSQP. The gap between the lower and upper bounds is expected to be tight because the gap is determined by coefficients of a single variable. The techniques used for strictly convex KSQP lead to achieve $O(n)$ time complexity of `CBS`, and new hybrid methods that are practically more reliable and faster are developed and tested experimentally using a broad range of random problems. Computations show that `CBS` is superior in speed and solution quality to the compared solvers such as Cplex 12.5. `CBS` also finds the exact global optima in some cases for which a characterization is given. In the experiments, the global optimum is found in 69% of instances when 30% of variables are related with concave terms.

5. **Practical applications of developed algorithms (Chapter 5)**

   An extensive list of applications for convex KSQP is presented. We demonstrate that mixed integer QP can be solved very efficiently with high solution quality through `CBS`. The superior speed and solution quality of the upper bounding procedure using `OBG` is verified by subset-sum problem with an available local solver.

1.4. Outline of this Dissertation

The general content of this dissertation are described in the following chapters.

Chapter 2: Knapsack Separable Quadratic Programming (KSQP): Survey

All existing methods of KSQP in convex and nonconvex cases are discussed, and ideas for computational enhancement are also proposed.

Chapter 3: Indefinite Open-Box Approach to Solution

A new global solver is developed for the indefinite case of KSQP, which has open box constraints ($l_j \leq x_j$ for concave terms). Although the algorithm, which is referred to Open Box Global (`OBG`) solver, finds an exact global optimum solution by enumerating all KKT points that satisfy an additional necessary condition, it has only $O(n^2)$ time worst case complexity. Furthermore, a new procedure that utilizes `OBG` to find lower and upper bounds for general KSQP is presented. The performances of developed algorithms are tested in experiments with local, global, and commercial solvers.

Chapter 4: Lagrangian-relaxed Closed-Box Solution

For indefinite KSQP with closed box constraints ($l_j \leq x_j \leq u_j$ for concave terms), a new $O(n)$ time tight bound algorithm, which is referred to Closed Box Solver (`CBS`), is developed. All root finding methods, which are discussed in the survey of Chapter 2 for strictly convex KSQP, are applied for `CBS`, and hybrid methods that consider efficiency and reliability are also proposed. All applied methods are computationally tested to select the best method in a broad range of random problems from strictly convex to highly concave KSQP. Then, it is compared with local, global, and commercial solvers. Impressive performance in speed and solution quality is achieved and discussed. The reason of high quality solution is explained by showing that the gap between lower and upper bound is determined by coefficients of a single variable.

Chapter 5: Applications

This chapter presents lists of the applications of convex KSQP. The real investment portfolio selection model is used to demonstrate that mixed integer QP can be solved using indefinite KSQP. Subset-sum problem, a direct application of indefinite KSQP is also discussed and tested.

Chapter 6: Concluding Remarks

The accomplishments and contributions of this dissertation are summarized, and potential directions for future research are discussed.

1.5. Notation and Preliminaries

1.5.1. Notation

- Bold lower case alphabet: a column vector (e.g. $\mathbf{a}$, $\mathbf{l}$, $\mathbf{u}$, $\mathbf{c}$, and $\mathbf{x}$)
- Bold upper case alphabet: a matrix (e.g. $\mathbf{Q}$ and $\mathbf{D}$)
- $\mathbf{0}$: bold 0 means a vector of zeros

- $\mathbf{1}$: bold 1 means a vector of ones
- n: the number of variables in a problem
- Superscript T of a vector or matrix means transpose (e.g. $\mathbf{x}^T \in \mathcal{R}^{1\times n}$ means the transpose of the vector $\mathbf{x}^T \in \mathcal{R}^{n\times 1}$.)
- Superscript $*$ of a scalar or vector means the optimum value (e.g. $\mathbf{x}^*$ means an optimum solution)
- $|\cdot|$ is the number of elements of the corresponding vector if the corresponding input is a vector or is the absolute value if the input is a scalar (e.g. $|\mathbf{a}| = 3$ for a vector $\mathbf{a} = [2, 4, 7]^T$, and $|-3| = 3$ for a scalar)
- $\lceil \cdot \rceil$ is the closest integer greater than the corresponding scalar. (e.t. $\lceil 1.3 \rceil = 2$)
- $\forall j$ means for all j (e.g. $\sum_{\forall j} a_j = \sum_{j=1}^{n} a_j$)
- x_L means x_j for $j \in L \in \{1, 2, ..., n\}$
- $[a, b]$: the interval of numbers between a and b including a and b (e.g. $c \in [a, b]$ means $a \leq c \leq b$). If the interval does not contain a or b, a parenthesis is used instead of the square bracket. (e.g. $c \in (a, b]$ means $a < c \leq b$)

1.5.2. Preliminaries[1]

A mathematical programming model is an optimization problem of the form

$$\begin{aligned} Min \quad & f(\mathbf{x}) \\ s.t. \quad & g_i(\mathbf{x}) \leq 0, \quad i = 1, ..., m \\ & \mathbf{x} \in \Omega \end{aligned} \tag{1.2}$$

where Ω is a nonempty subset of $\mathcal{R}^n$ and is in the domain of the real-valued functions, $f : \mathcal{R}^n \to \mathcal{R}^1$, and $g_i : \mathcal{R}^n \to \mathcal{R}^1$ for $i = 1, ..., m$. The relation, $g_i(\mathbf{x}) \leq 0$ is called a

[1]This part mainly refers to [10, 141] so citation for specific parts are omitted in this section.

constraint, and $f(\mathbf{x})$ is called the objective function.

A given $\mathbf{x} \in \mathcal{R}^n$ is feasible if it is in the domain of Ω and satisfies the constraints $g_i(\mathbf{x}) \leq 0$. A point $\mathbf{x}^*$ is said to be a global optimum if it is feasible and if the value of the objective function is not more than that of any other feasible solution: $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all feasible $\mathbf{x}$. A point $\hat{\mathbf{x}}$ is said to be a local optimum if there exists an ε-neighborhood $N_\varepsilon(\hat{\mathbf{x}})$, i.e., a ball of radius ε with center at $\hat{\mathbf{x}}$, such that $f(\hat{\mathbf{x}}) \leq f(\mathbf{x})$ for all feasible $\mathbf{x} \in N_\varepsilon(\hat{\mathbf{x}})$.

A set S is said to be convex if the line segment connecting any two points in the set belongs to the set. That is, if $\mathbf{x}_1$ and $\mathbf{x}_2$ are any two points in the set S, then a linear combination of these two points, denoted by $\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2$, is in S for any $\lambda \in [0,1]$. A function f is said to be convex on S if $f[\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2] \leq \lambda f(\mathbf{x}_1) + (1-\lambda)f(\mathbf{x}_2)$, for each $\mathbf{x}_1$ and $\mathbf{x}_2$ in S and for each $\lambda \in [0,1]$.

The Karush-Kuhn-Tucker (KKT) necessary optimality conditions can be stated as

$$\begin{array}{rcc} \textit{Stationarity} & \nabla_{\mathbf{x}} f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla_{\mathbf{x}} g_i(\mathbf{x}) = 0 & \\ \textit{Primal feasibility} & g_i(\mathbf{x}) \leq 0,\ \mathbf{x} \in \Omega & i = 1, ..., m \\ \textit{Complementary slackness} & \lambda_i g(\mathbf{x})_i = 0 & i = 1, ..., m \\ \textit{Dual feasibility} & \lambda_j \geq 0 & i = 1, ..., m \end{array}$$

where $\nabla_{\mathbf{x}} f$ represents the gradient vector with respect to $\mathbf{x}$, and λ_i is the Lagrange multipliers associated with the constraints $g_i(\mathbf{x}) \leq 0$. If $\mathbf{x}^*$ is a local minimum for the problem in (1.2) and constraint qualification holds at $\mathbf{x}^*$, then $\mathbf{x}^*$ satisfies the KKT conditions. In addition, if $f(x)$ and $g_i(\mathbf{x})$ are differentiable and convex, $\mathbf{x}^*$ is a global minimum solution to the problem in (1.2) if $\mathbf{x}^*$ satisfies the KKT conditions.

Let the given problem in (1.2) be the primal problem. Then, there exists a problem that is closely associated with it, called the Lagrangian dual problem in (1.3).

$$Max_{\boldsymbol{\lambda} \geq \mathbf{0}}\ \theta(\boldsymbol{\lambda}) \tag{1.3}$$

where $\theta(\boldsymbol{\lambda}) = inf\{f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla g_i(\mathbf{x}) : \mathbf{x} \in \Omega\}$, and λ_i in $\boldsymbol{\lambda} \in \mathcal{R}^m$ is referred to dual variable or Lagrange multiplier of the corresponding $g_i(\mathbf{x}) \leq 0$. It may be noted that $\theta(\boldsymbol{\lambda})$

is a concave function even in the absence of any convexity or concavity assumptions on f or g, or convexity of the set Ω. If $\mathbf{x}$ is feasible to the primal problem (1.2) and if $\boldsymbol{\lambda} \geq \mathbf{0}$, then $f(\mathbf{x}) \geq \theta(\boldsymbol{\lambda})$. This result is called the weak duality theorem. Suppose that Ω is convex and that f, $g_i : \mathcal{R}^n \to \mathcal{R}^1$ for $i = 1, ..., m$ are convex. Then, the objective values of both primal and dual problems are equal. This result is called the strong duality theorem.

2. Knapsack Separable Quadratic Programming (KSQP): Survey

2.1. Introduction

Consider a continuous knapsack separable quadratic program (KSQP)

$$\begin{aligned} \text{(KSQP)} \quad Min \quad & \frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{c}^T\mathbf{x} \\ s.t. \quad & \mathbf{a}^T\mathbf{x} = b \\ & \mathbf{l} \le \mathbf{x} \le \mathbf{u} \end{aligned} \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is decision variables, $\mathbf{c}, \mathbf{a}, \mathbf{l}, \mathbf{u} \in \mathbb{R}^n$, $b \in \mathbb{R}$, and $\mathbf{D} \in \mathbb{R}^{n \times n}$ is a diagonal matrix with its diagonal elements d_j. Without loss of generality, we assume $l_j < u_j$ and $a_j \neq 0$ for $\forall j$. In the case of $a_j = 0$, the optimum solution for the variables is trivial. See Appendix A.1 for detail.

Transformation

Throughout this dissertation, three index sets

$$P = \{j : d_j > 0\}, \quad Z = \{j : d_j = 0\}, \quad N = \{j : d_j < 0\}$$

are used, and KSQP is considered in the form of (P):

$$\begin{aligned} (P) \quad Min \quad & \frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{c}^T\mathbf{x} \\ s.t. \quad & \sum_{j\in\forall j} x_j = b \\ & \mathbf{l} \le \mathbf{x} \le \mathbf{u} \end{aligned} \tag{2.2}$$

such that

$$c_j = 0 \ for \ j \notin Z.$$

The form of (P) can be obtained from any KSQP through the transformation of

$$x_j \leftarrow \begin{cases} x_j/a_j + c_j/d_j & for \ j \notin Z \\ x_j/a_j & for \ j \in Z \end{cases}. \tag{2.3}$$

Appendix A.2 explains the detail of the transformation, and the computational benefit of the transformation is discussed in Appendix A.3.

KKT conditions

KKT conditions of KSQP are essential conditions to derive our algorithms and explain the existing algorithms. KKT conditions of (P) are

$$\left.\begin{array}{rcc} Stationarity & d_jx_j - c_j + \lambda + \mu_j - \gamma_j = 0 & \forall j \\ Primal \ feasibility & \sum_{\forall j} x_j = b, \ l_j \le x \le u_j & \forall j \\ Complementary \ slackness & \mu_j(x_j - u_j) = 0, \ \gamma_j(l_j - x_j) = 0 & \forall j \\ Dual \ feasibility & \mu_j \ge 0, \ \gamma_j \ge 0 & \forall j \end{array}\right\} \tag{2.4}$$

where λ is a Lagrange multiplier for the knapsack constraint, $\sum_{j\in\forall j} x_j = b$, and μ_j and γ_j are Lagrange multipliers of upper and lower bounds of each variable.

Strictly convex case

If all d_j's are strictly positive, i.e. $d_j > 0 \ \forall j$, then KSQP is a strictly convex problem. Since every KKT point is the global optimum solution in this case, according to the strong duality theorem, the problem is relatively easy so $O(n)$ time complexity algorithms have been developed. Strictly convex KSQP has a lot of applications and is mainly used as a sub-problem to project the current solution onto the feasible domain. Much research and various algorithms have been developed since 1963 as summarized in Table 2.1.

Table 2.1.: History of Algorithms for strictly convex case

Methods	Complexity	1960's	1970's	1980's	1990's	2000's	2010-2013	Total
Bisection	$O(n\, log\, n)$	1963 [123]	[116]	[93, 94, 95, 27, 36, 117, 133]	[120, 77, 137]		[140] 2012	13
Sorting (Ranking)	$O(n\, log\, n)$	1969 [29]	[49, 57, 75, 61]	[50, 2, 143, 39, 135, 112, 28]	[134, 88, 18] 1995			15
Pegging (Var. fixing)	$O(n^2)$		1971 [116, 75]	[143, 14, 121, 82, 54, 133]	[134, 111, 19, 15]	[124, 125, 126, 17, 127, 128, 65]	[62] 2012	20
Median search	$O(n)$			1984 [20, 21, 76]	[101, 134, 26, 52, 77]	[78, 63, 64]	[32] 2013	12
Secant	Not measurable			1988 [112, 117]	[137]	[31]	[25]2012	5
Newton	$O(n^2)$				1992 [88, 111, 137]	[74]	[25] 2012	5
Interval test	$O(n\, log\, n)$				1992 [132]		[7, 8] 2013	3
Interior point	Not researched						[136] 2013	1
Total		2	7	25	20	12	8	74

1. Citations are ordered by publication year.
2. The year of initial (last) paper is presented before (after) the corresponding citation.

Positive semidefinite case

If $\mathbf{D}$ is positive semidefinite with diagonal elements of $d_j \geq 0 \; \forall j$, KSQP is a convex problem. Typically, QP with positive semidefinite matrix for quadratic terms is considered to be more difficult than the strictly convex case so the problem is called ill-conditioned problem. However, because convex KSQP still satisfies the strong duality theorem and separable with a diagonal matrix $\mathbf{D}$, its all global optimum solutions are quite easily derived from KKT conditions. Chapter 4 explains this derivation.

Nonconvex case

In the case of $d_j < 0$ for some j, problem KSQP is notoriously difficult, and Pardalos and Vavasis ([104]) proved that it is an *NP-hard* problem even if only one d_j is negative. Although it is theoretically known that the global optimum solution is at a boundary point allowing at most one x_j to be strictly within (l_j, u_u) (see Theorem 4.8), no efficient method has been developed in the literature.

Due to such difficulty, the investigation for nonconvex case of KSQP has been rarely considered. In this author's best knowledge, there exist only four studies in [83, 101, 131, 132] which find local or approximate solutions, and performances of these methods deteriorate significantly as the number of negative d_j increases.

A comprehensive literature review of all convex and nonconvex cases of KSQP is presented in this chapter with theoretical and computational analysis, and ideas that improve the existing methods are also suggested. Hybrid methods that combine the existing algorithms for convex KSQP are discussed in Section 4.3.1, and experimental results for the strictly convex KSQP algorithms, including hybrid methods, are also presented in Section 4.4.1.

2.2. Strictly Convex case

Survey papers of Bretthauer et al. [16] and Patriksson [106] classified methods into two types: Lagrange multiplier search method (also called break point search method or relaxation method) and pegging method (also called variable fixing method). The main difference between the two methods is the way to approach finding the optimum solution. The Lagrange multiplier search method searches for the optimum Lagrange multiplier violating the knapsack constraint ($\mathbf{a}^T\mathbf{x} = b$), while the pegging method iteratively finds a solution violating the bound constraints ($\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$) until all variables are within the bound. Patriksson [106] categorized the Lagrange multiplier search method as explicitly dual and implicitly primal and pegging method as explicitly primal and implicitly dual.

The Lagrange multiplier search methods that we present in this chapter are bisection, sorting (ranking), exact median search (binary search), approximate median search, secant, Newton (Quasi Newton, semi smooth Newton, dual Newton), and interval test methods. Other methods we consider are pegging and interior point methods. For these, see Table 2.1.

Besides the above methods, Nielsen and Zenios (1992, [88]) also applied methods of Bregman projection and Tseng [130] on strictly convex KSQP. Bregman projection method is similar to the pegging method, and Tseng [130] finds the optimum solution underestimating the optimal step. However, we do not consider those two methods because (a) Bregman projection method does not converge finitely, (b) [88] concluded that Newton method is more robust than the two methods.

The review of this section is based on the review in Patriksson (2008, [106]), which focuses on the problems minimizing separable, convex, and differentiable objective function with a single separable convex knapsack constraint and box constraints in the continuous space including the strictly convex case of (P). Hence, this chapter is a successor of Patriksson [106] focusing on the linear knapsack quadratic problem (P).

Ventura (1991, [134]) and Robinson et al. (1992, [111]) also give an overview of the methods that were available at that time and conducted experiments to compare with their own methods. Experimental results from literature are summarized and compared in Section 2.2.4 with a summary Table 2.5. While reviewing the existing methods, we also present computationally improved Newton methods.

2.2.1. Solution characteristics

This section briefly describes the solution characteristics in order to explain the existing algorithms. Most existing algorithms of convex KSQP find an optimum solution utilizing KKT conditions, e.g. [18, 50, 64]. From the KKT conditions in (2.4), the following results are derived.

$$\left.\begin{array}{ll} \text{If } x_j = l_j, & \text{then } \mu_j = 0 \text{ and } \lambda \geq -d_j l_j \\ \text{If } x_j = u_j, & \text{then } \gamma_j = 0 \text{ and } \lambda \leq -d_j u_j \\ \text{If } x_j = (l_j, u_j), & \text{then } \mu_j = \gamma_j = 0 \text{ and } \lambda = -d_j x_j \end{array}\right\} \tag{2.5}$$

Then, (2.5) leads to define the following solution characteristics with a given λ as

$$x_j(\lambda) = median\{l_j, u_j, -\lambda/d_j\}. \tag{2.6}$$

Since $x_j(\lambda)$ satisfies all KKT conditions except for the knapsack constraint $\sum_{\forall j} x_j = b$, most existing algorithms focus on finding the optimum λ^* that solves the equation

$$g(\lambda) \equiv \sum_{\forall j} x_j(\lambda) - b = 0.$$

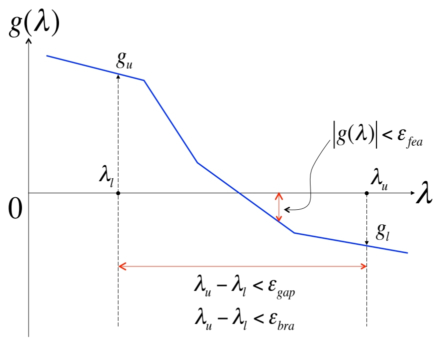
Since $x_j(\lambda)$ is a piecewise nonincreasing function having breakpoints $-d_j l_j$ and $-d_j u_j$ on λ, $g(\lambda)$ also inherits the same properties. Thus, the existing algorithms mainly focus on searching for λ^*, a root of $g(\lambda)$, i.e. λ^* such that $g(\lambda^*) = 0$.

2.2.2. Algorithms

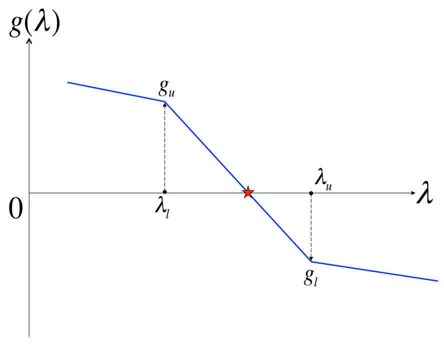
This section introduces the algorithms for strictly convex case of (P). We define

$$\begin{aligned} \lambda_{min} &= min\{-d_j u_j : \forall j\} \\ \lambda_{max} &= max\{-d_j l_j : \forall j\}, \end{aligned}$$

and two values λ_l and λ_u are used such that $\lambda_l \leq \lambda_u$ and $g_l \leq 0 \leq g_u$, where $g_l = g(\lambda_u)$ and $g_u = g(\lambda_l)$. Figure 2.1 illustrates examples.


$g(\lambda)$
g_u
$|g(\lambda)| < \varepsilon_{fea}$
λ_l
λ_u
λ
0
g_l
$\lambda_u - \lambda_l < \varepsilon_{gap}$
$\lambda_u - \lambda_l < \varepsilon_{bra}$

(a) Tolerances



(b) Interpolation

Figure 2.1.: λ_l, λ_u, g_l, g_u, and tolerances

We also define four positive tolerances:

$$\begin{array}{lcl} \varepsilon_{gap} & > & \lambda_u - \lambda_l \\ \varepsilon_{fea} & > & |g(\lambda)| \\ \varepsilon_{bra} & > & \lambda_u - \lambda_l \\ \varepsilon_{peg} & = & \varepsilon_{fea} \end{array}, \tag{2.7}$$

which are used to stop iteration or ensure convergence, so inequalities in (2.7) are the termination conditions.

As Figure 2.1a, ε_{gap} is used to decide whether λ_l and λ_u are close enough to conclude λ^* is placed within a tolerance range. ε_{bra} is used to give a range $[\lambda_l, \lambda_u]$ as a bracket to interval test method so it may generally be greater than ε_{gap}. ε_{fea} determines the level of feasibility of the knapsack constraint measuring the violation of $g(\lambda)$ from *zero*, and ε_{peg} is used by pegging method to determine feasibility so its role is like that of ε_{fea}.

As observed in Figure 2.1, λ^* can be found by any root finding method[1] such as golden section and Fibonacci search. But we present root finding methods that have been considered in the literature for KSQP. Furthermore, two methods (interval test and pegging methods) that do not use root finding methods are also discussed.

[1]See a book, "Numerical Mathematics" by Quarteroni et al. [109] for general root finding methods.

Bisection method

The Bisection method is most intuitive, and it was first applied by Srikantan (1963, [123]). The method has been studied by [123, 116, 93, 94, 95, 27, 36, 117, 133, 120, 77, 137, 140] in the order of publication years from 1963 to 2012. Its complexity is $O(n\ log\ n)$ and is well explained in the recent paper of Zhang and Hua (2012, [140]).

The biggest advantage of the bisection method is its guarantee of convergence because the range of λ is halved at every iteration. If the method terminates satisfying

$$\lambda_u - \lambda_l < \varepsilon_{gap}, \tag{2.8}$$

the maximum number of iterations is $\left\lceil log_{0.5}\left(\frac{\varepsilon_{gap}}{\lambda_{max}-\lambda_{min}}\right)\right\rceil$, since the method iterates until $(\lambda_{max} - \lambda_{min})0.5^{Max.Iter.} < \varepsilon_{gap}$ is satisfied.

However, if the initial gap between λ_{max} and λ_{min}, which are generated by the given problem, is large, the expected number of iterations increases. Moreover, the termination criterion (2.8) with ε_{gap} does not guarantee feasibility. For example, if multiple different breakpoints are within the last $[\lambda_l, \lambda_u]$, whose range is less than ε_{gap}, the bisection method may finish its iteration with an infeasible $\mathbf{x}(\lambda^*)$. Therefore, the bisection method should use the termination criterion

$$|g(\lambda)| < \varepsilon_{fea}$$

to guarantee a feasible solution. A schema for the bisection method with the more reliable termination criterion is shown in Algorithm 2.1.

Sorting method

The Sorting method has been researched extensively in [29, 49, 57, 75, 61, 50, 2, 143, 39, 135, 112, 28, 134, 88, 18] since it was first implemented by Dafermos and Sparrow (1969, [29]). Most of the literature generally cite and implement Helgason et al. (1980, [50]). We name the algorithm "the sorting method" in this dissertation. Note that Robinson et al.

Algorithm 2.1 $O(n\ log\ n)$; Bisection method

```
1. Set $\lambda_l = \lambda_{min}$, $\lambda_u = \lambda_{max}$, and $\lambda = (\lambda_l + \lambda_u)/2$
2. Iterate while $|g(\lambda)| \geq \varepsilon_{fea}$
    a) If $g(\lambda) < 0$, set $\lambda_u = \lambda$
       Else set $\lambda_l = \lambda$
    b) Set $\lambda = (\lambda_l + \lambda_u)/2$
3. Finish with $\lambda^* = \lambda$ and $\mathbf{x}(\lambda^*)$
```

[111] called the method bisection search method.

The complexity of the sorting method is $O(n\ log\ n)$. Most literature use QUICKSORT for their implementation and experiments, but HEAPSORT and MERGESORT have been used by Bretthauer [18] and Ventura [134] respectively. The three sorting algorithms have $O(n\ log\ n)$ complexity on average, but QUICKSORT has $O(n^2)$ complexity in the worst case, while other two algorithms keep $O(n\ log\ n)$ complexity in the worst case. Helgason et al. [50] gives a worst case complexity analysis of operations for this method.

The sorting step is the main cost of the sorting method. According to our experiments, about 30%[2] of the total time is spent to sort $2n$ breakpoints. For this reason, alternative methods such as median search method have been developed to avoid sorting steps, and the sorting method has not been used since Bretthauer and Shetty (1995, [18]). Moreover, Bretthauer and Shetty switched to the pegging method for their succeeding studies in [15, 17, 19].

Algorithm

A scheme of sorting method is presented in Algorithm 2.2. It initially sorts all breakpoints at step 1 and reduces breakpoints by half considering the sign of $g(T_j)$ where T_j is the median breakpoint among remaining and sorted breakpoints. This iterative step makes sorting method guarantee to converge at most $\lfloor 1 + log_2(2n) \rfloor$ iterations[3], which is fixed by

[2]MATLAB provides execution time for each line through a function `profiler.m`.

[3]The maximum iteration can be obtained from $1 \geq 2n/2^{Max.Iter.-1}$

n. In step 2.a, if the number of remaining breakpoints, $|T|$, is an even number, one can also update λ by the mid point of the two middle breakpoints depending on the definition of median.

Algorithm 2.2 $O(n\ log\ n)$; Sorting method

```
1. Get a breakpoint set $T = \{-d_j l_j, -d_j u_j\}$
2. Sort $T = sort(T)$ in ascending order
3. Iterate while $T \neq \emptyset$ (all break points are excluded)
    a) Get $\lambda = T_j$ for $j = \lceil |T|/2 \rceil$
    b) If $|g(\lambda)| < \varepsilon_{fea}$, finish algorithm with $\lambda^* = \lambda$ and $\mathbf{x}(\lambda^*)$
       Else if $g(\lambda) < 0$, set $\lambda_u = \lambda$ and $g_l = g(\lambda)$, exclude break points by $T = T \backslash \{T_j : T_j \geq \lambda\}$
       Else set $\lambda_l = \lambda$ and $g_u = g(\lambda)$, exclude break points $T = T \backslash \{T_j : T_j \leq \lambda\}$
4. Interpolate to get $\lambda^* = \lambda_l + g_u(\lambda_u - \lambda_l)/(g_u - g_l)$ and finish algorithm with $\mathbf{x}(\lambda^*)$
```

Note that, in step 3.a, we just use $j = \lceil |T|/2 \rceil$ for median index, but median of T can be defined $(T_j + T_{j+1})/2$ if $|T|$ is an even number by the definition of median.

Remark 2.1. It is worth noting that Patriksson [106] made the error of classifying the classic sorting method literature of Helgason et al. [50] as a bisection method. Maybe due to the reason, Patriksson [106] also erroneously classified Bretthauer et al. [18] and Ventura [134], which stemmed from [50], as a bisection method.

Exact median search method (Binary search)

The exact median search method, which is also called as binary search, uses a selection algorithm to pick the median breakpoint instead of initially sorting all breakpoints as in the sorting method. The replacement makes the exact median search method be more efficient than the sorting method thanks to $O(n)$ time selection algorithms such as SELECT of [66]. Exact median search method converges in less than $\lfloor 1 + log_2(2n) \rfloor$ iterations, which is same as the sorting method, but theoretical complexity is reduced to $O(n)$ time. For detailed complexity analysis, see [64, 78].

Since this method was initially developed by Brucker (1984, [20]), numerous researchers [20, 21, 76, 26, 52, 77, 78, 63, 64, 32] have improved the method, and it seems Kiwiel (2008, [64]) finalized the research of the median search methods with his modified version showing, with counter examples, that previous methods [26, 52, 77, 76, 101] may cycle. The result of his experiments which compare median search methods is summarized in the Section 2.2.4.

A conceptual procedure of the exact median search method is available in Algorithm 2.3 and a more efficient and sophisticated version is provided in Kiwiel [64].

Algorithm 2.3 $O(n)$; Exact Median search method (Binary Search)

```
1. Get a breakpoint set $T = \{-d_j l_j, -d_j u_j\}$
2. Iterate while $T \neq \emptyset$ (all break points are excluded)
    a) Get $\lambda = median(T)$
    b) If $|g(\lambda)| < \varepsilon_{fea}$, stop algorithm with $\lambda^* = \lambda$ and $\mathbf{x}^* = \mathbf{x}(\lambda^*)$
       Else if $g(\lambda) < 0$, set $\lambda_u = \lambda$ and $g_l = g(\lambda)$, exclude break points by $T = T \backslash \{T_j : T_j \geq \lambda\}$
       Else set $\lambda_l = \lambda$ and $g_u = g(\lambda)$, exclude break points $T = T \backslash \{T_j : T_j \leq \lambda\}$
3. Interpolate to get $\lambda^* = \lambda_l + g_u(\lambda_u - \lambda_l)/(g_u - g_l)$ and finish with $\mathbf{x}(\lambda^*)$
```

Approximate median search method (Random search)

The approximate median search method was developed by Pardalos and Kovoor (1990, [101]). Ventura (1991, [134]) suggested a similar idea with the name "random search" in order to avoid sorting in [50]. The approximate median search method is very similar to the exact median search method. It randomly chooses one remaining breakpoint instead of picking the exact median of breakpoints. So the method can be presented replacing step 2.a of the exact median search method in Algorithm 2.3 with Algorithm 2.4 keeping the other steps the same.

This simple idea makes the method have expected $O(n)$ time complexity with a small time constant removing the exact median search procedure; however, its complexity grows

Algorithm 2.4 Expected $O(n)$; Approximate median search method (Random search)

```
This algorithm is completed by replacing step 2.a in Algorithm 2.3 with the following step.
  2. Iterate while $T \neq \emptyset$ (all break points are excluded)
      a) Get $\lambda = r$, where $r$ is a randomly chosen element in $T$
```

to $O(n^2)$ in the worst case.

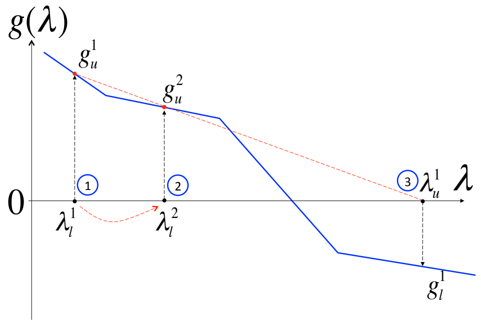
Secant method

The secant method was first proposed by Rockafellar and Wets (1988, [112]) for strictly convex KSQP. In spite of its competitive performance, it has been less frequently considered by [117, 137, 31, 25] (in the order of their publication year) than others because a well developed method was proposed recently by Dai and Fletcher (2006, [31]).

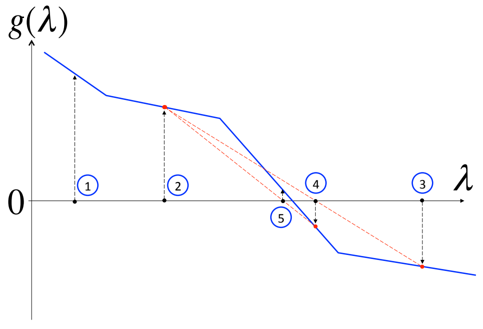
The secant method of [31] uses a variant of false position (regula falsi) root finding method, which is also a variant of the secant root finding method. False position method may be less rapid than secant method but is safer because secant method travels out of the initial bracket of $[\lambda_{min}, \lambda_{max}]$ [108]. Because $g(\lambda)$ is a piecewise non-increasing linear function, the false position method can be intuitively applied enjoying the properties of $g(\lambda)$. Because $g(\lambda)$ is continuous everywhere with linear pieces if (P) is in strictly convex case, the final secant step finds the exact λ^*.

Algorithm

The secant method of Dai and Fletcher (2006, [31]) has two phases as usual root finding methods: bracketing phase and secant phase. Bracketing phase is used to find a bracket $[\lambda_l, \lambda_u]$ such that $g(\lambda_l) \geq 0$ and $g(\lambda_u) \leq 0$. At first, it calculates two $g(\lambda)$'s with a given initial λ and updated λ with a given step and finds the third λ as Figure 2.2a if $g(\lambda)$ with the second λ has the same sign of $g(\lambda)$ as the initial λ. Then, it iterates until it finds a bracket with a safeguard to avoid too small steps. The safeguard is used in step 2.b.ii and step 2.e.ii of Algorithm 2.5. So the step size $\Delta\lambda$ is updated by $\Delta\lambda = \Delta\lambda + \Delta\lambda/s$ in a way like secant



(a) Bracketing phase



(b) Secant phase

Figure 2.2.: Secant method

root finding method, and it restricts s to be at least 0.1 as in (2.10) to ensure that a new step size is at least 10 times the previous step size. Dai and Fletcher [31] presented a pseudo code for the bracket phase and its line 6 and 15 are related to this; however, two lines are not correct. For example, the line 6 in our notation is given in (2.9), and it is trivial that the value of s is always 0.1 because g_u is updated before s is computed. So we communicated with two authors, and they accepted the error on April 24, 2013. So we corrected it in (2.10) in our implementation.

$$Error\ code: \quad \lambda_l = \lambda; g_u = g(\lambda); s = max(g_u/g(\lambda) - 1, 0.1); \tag{2.9}$$

$$Correct\ code: \quad \lambda_l = \lambda; s = max(g_u/g(\lambda) - 1, 0.1); g_u = g(\lambda); \tag{2.10}$$

Once we obtain a bracket, secant phase is applied to find the optimum solution as Figure 2.2b. The secant phase basically uses the false position method but a modification is made to avoid too small reduction ensuring that the range is reduced by a factor of 3/4 or less in step 2.a.ii and step 2.a.iv of Algorithm 2.6.

Algorithm 2.5 Bracketing phase: Find a bracket $[\lambda_l, \lambda_u]$

```
1. Give initial $\lambda$ and step size $\triangle\lambda$ and get $g(\lambda)$
2. If $g(\lambda) > 0$
    a) Update $\lambda_l = \lambda$, $g_u = g(\lambda)$, $\lambda = \lambda + \Delta\lambda$ and get new $g(\lambda)$
    b) Iterate while $g(\lambda) > 0$
        i. Set $\lambda_l = \lambda$
        ii. Update $\triangle\lambda$ and $\lambda$ using the modified secant method and set $g_u = g(\lambda)$
        iii. Get new $g(\lambda)$
    c) Set $\lambda_u = \lambda$ and $g_l = g(\lambda)$
   Else ($g(\lambda) \leq 0$)
    d) Update $\lambda_u = \lambda$, $g_l = g(\lambda)$, $\lambda = \lambda - \Delta\lambda$ and get new $g(\lambda)$
    e) Iterate while $g(\lambda) < 0$
        i. Set $\lambda_u = \lambda$
        ii. Update $\triangle\lambda$ and $\lambda$ using the modified secant method and set $g_l = g(\lambda)$
        iii. Get new $g(\lambda)$
    f) Set $\lambda_l = \lambda$ and $g_u = g(\lambda)$
```

Pegging (Variable fixing, Projection algorithm)

The pegging algorithm is widely used for strictly convex resource allocation problems with a single knapsack constraint and bounds. One can see the book by Ibaraki and Katoh [54] and a review by Patriksson [106] for nonlinear resource allocation problems with pegging method. According to Patriksson [106], the idea of pegging algorithm was presented by Sanathanan (1971, [116]), and Bitran and Hax (1981, [14]) is the true pegging algorithm with convergence theory. So most variant methods have been developed based on Bitran and Hax (1981, [14]) in [19, 65, 62, 127, 134] for problem (P). As we can see in the algorithm history Table 2.1, pegging method is one of the more frequently studied algorithms in recent years for (P). It is due to its outstanding performance compared with Lagrange multiplier search methods and its flexibility for nonlinear functions. Other literature that covers (P) are [116, 75, 143, 14, 121, 82, 54, 133, 111, 19, 15, 124, 125, 126, 17, 127, 128, 65, 62] in chronological order.

Algorithm 2.6 Secant method: Do secant update

```
With given $\lambda$, $\Delta\lambda$, $\lambda_l$, and $\lambda_u$ from Bracketing pahse
  1. Update $\triangle\lambda$ and $\lambda$ using the secant method
  2. Iterate while $|g(\lambda)| \geq \varepsilon_{fea}$
     a) If $g(\lambda) < 0$, update $\lambda_u = \lambda$ and $g_l = g(\lambda)$
          i. If a new $\lambda$ is on the left half side of the bracket, update $\lambda$ by secant
         ii. Else set a new $\lambda$ that reduces the bracket size by 3/4 or less.
        Else update $\lambda_l = \lambda$ and $g_u = g(\lambda)$
        iii. If a new $\lambda$ is on the right half side of the bracket, update $\lambda$ by secant
         iv. Else set a new $\lambda$ that reduces the bracket size by 3/4 or less.
     b) Get new $g(\lambda)$
  3. Finish with $\lambda^* = \lambda$ and $\mathbf{x}(\lambda^*)$
```

Algorithm

Recently Kiwiel (2008, [65]) did a comprehensive review for pegging method and addressed an improved and modified version proving, with counter examples, that [19, 111] may fail. We present a new version of pegging method in Algorithm 2.7, which is simpler than the version of Kiwiel[65] with fewer index sets.

Pegging method is conceptually different from other Lagrange multiplier search methods. As we will show the difference with a geometric interpretation in Section 2.2.3, pegging method does not search for λ^*, instead it iteratively projects (step 2) the center of the objective function to the hyperplane of the knapsack constraint and pegs (fixes) (step 3.c) a part of variables at its bound l_j or u_j dependent on the violation of the projected point to the box constraint. For these reasons, pegging method is also called (restricted) projection method (Robinson et al. [111]) or variable fixing method (Kiwiel [65]).

The minimum solution $\mathbf{x}$ in step 2 geometrically means projecting the origin of the objective function to the hyperplane (knapsack constraint) proportional to $1/d_j$ in the space of unpegged variable index set I. We may get the projected point $\mathbf{x}$ quite easily using KKT conditions ignoring box constraint. From the KKT stationarity condition (2.4), we

Algorithm 2.7 $O(n^2)$; Pegging method (Variable fixing)

```
1. Let $I = \{1, 2, ..., n\}$
Iterate until stopping criteria is satisfied
2. Solve a sub-problem (Projection)
   $\mathbf{x} = argmin\{\sum_{j\in I} d_j x_j^2 : \sum_{j\in I} x_j = b\}$
3. Feasibility check
   a) Get infeasibility indicator (Force $\mathbf{x}$ into box)
      i. Get $S_L = \sum_{j\in L} l_j$ and $\nabla = S_L - \sum_{j\in L} x_j$ for $L = \{j \in I : x_j \leq l_j\}$
      ii. Get $S_U = \sum_{j\in U} u_j$ and $\Delta = \sum_{j\in U} x_j - S_U$ for $U = \{j \in I : x_j \geq u_j\}$
   b) Stopping criterion
      i. If $|\nabla - \Delta| < \varepsilon_{peg}$, finish with $x_j = l_j$ for $j \in L$ and $x_j = u_j$ for $j \in U$
   c) Pegging (Variable fixing)
      i. If $\nabla > \Delta$, then $I = I \backslash L$, $b = b - S_L$, and $x_j = l_j$ for $j \in L$
      ii. If $\nabla < \Delta$, then $I = I \backslash U$, $b = b - S_U$, and $x_j = u_j$ for $j \in U$
```

can derive

$$d_j x_j + \lambda = 0 \rightarrow x_j = \frac{-\lambda}{d_j} \tag{2.11}$$

and substitute (2.11) into the knapsack constraint to get a closed form of λ as in (2.12).

$$\sum_{j\in I} \frac{-\lambda}{d_j} = b \rightarrow \lambda = \frac{-b}{\sum_{j\in I} 1/d_j} \tag{2.12}$$

Therefore, we can get the solution at step 2 as

$$x_j = \left(\frac{b}{\sum_{j\in I} 1/d_j}\right) / d_j \quad for\ j \in I \tag{2.13}$$

This simple closed form to get $\mathbf{x}$ is the source of advantage of pegging method. Patriksson [106] concluded that pegging method generally performs better than Lagrange multiplier search methods if a problem has a closed form to get its projection onto the knapsack constraint.

Another efficiency source is the fact that the number of variables to be projected in step

2 decreases with every iteration, as it will be proved in proposition 2.2. Once the projected point is obtained, we consider the box constraint ($\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$) in step 3. Robinson et al. [111] named this step "force $\mathbf{x}$ into box." Then, we check the feasibility of the box forced point on the knapsack constraint in step 3.c. If it is not feasible, we can peg some variables at its extreme points. For example, consider we have $\bar{\lambda}$ and it results in $g(\bar{\lambda}) > 0$ and $x_j(\bar{\lambda}) = l_j$ for some j. Then, since $g(\lambda)$ is non-increasing, it holds $\bar{\lambda} < \lambda^*$, and $x_j(\lambda)$ stays at l_j for any $\bar{\lambda} \leq \lambda$ because it is also a non-increasing function (see Figure 4.1a for intuition). Similarly, the case of $\lambda^* > \bar{\lambda}$ with $g(\bar{\lambda}) < 0$ and $x_j(\bar{\lambda}) = u_j$ is trivial. Therefore, we can peg x_j^* by

$$x_j^* = \begin{cases} l_j & for\ j \in L = \{j : x_j(\lambda) = l_j, \} \quad if\ g(\lambda) > 0, \\ u_j & for\ j \in U = \{j : x_j(\lambda) = u_j, \} \quad if\ g(\lambda) < 0, \end{cases} \tag{2.14}$$

The two steps of box forcing and pegging in step step 3.a and step 3.c can be efficiently done with two variables: up triangle Δ and down triangle ∇. Consider the initial steps assuming all variables are unpegged, that is, $I = \{1, 2, ..., n\}$. Then, the violation of knapsack constraint $g(\lambda)$ with $\mathbf{x}$ in (2.13) can be expressed by Δ and ∇ with

$$\begin{aligned} g(\lambda) &= \sum_{\forall j} x_j(\lambda) - b \\ &= \sum_{j \in I} x_j - b + S_L - \sum_{j \in L} x_j + \sum_{j \in U} x_j - S_U \quad since \sum_{j \in I} x_j = b \\ &= \nabla - \Delta \end{aligned} \tag{2.15}$$

where $L = \{j \in I : x_j \leq l_j\}$, $U = \{j \in I : x_j \geq u_j\}$, $\nabla = S_L - \sum_{j \in L} x_j$, and $\Delta = \sum_{j \in U} x_j - S_U$. This leads to prove proposition 2.2.

Proposition 2.2. *The pegging method in Algorithm 2.7 pegs at least one variable per iteration so it converges in at most n iterations.*

Proof. It is easily verified that

$$g(\lambda) > 0 \quad if\ and\ if\ only \quad \nabla > \Delta$$
$$g(\lambda) < 0 \quad if\ and\ if\ only \quad \nabla < \Delta$$

by (2.15). Thus, if $\Delta \neq \nabla$, since $g(\lambda) \neq 0$ at least one x_j should be pegged at one of its bounds l_j and u_j by (2.14). Therefore, the maximum iteration is n if we assume only one x_j is pegged every iteration as the worst case, and the stopping criterion in step 3.b guarantees the optimum solution $\mathbf{x}^*$ because it satisfies all KKT conditions, which are necessary and sufficient conditions. □

Other good approaches for convergence proofs are also available in [65, 111]. The proposition 2.2 is also a clue to get the complexity of pegging method. All steps in the Algorithm 2.7 have $O(n)$ time, but the total complexity is $O(n^2)$ because it can iterate n times in the worst case. The computational cost in the worst case of iteration part step 2-3 is $4n^2 + 7n$ as computed in Table 2.4. In spite of the quadratic complexity, pegging method may be efficient in practice in the sense that it avoids median search or sorting, and it is proved by previous studies as in experiment history Table 2.5.

Kiwiel [65] noticed two important modifications from previous methods. The first one is a way to get L and U in step 3.a. It finds index sets that contain the case of $x_j = l_j$ or u_j, while Robinson et al. [111] and Bretthauer et al. [19] define $L = \{j \in I : x_j < l_j\}$ and $U = \{j \in I : x_j > u_j\}$ with strict inequalities. This modification is simple but can accelerate pegging method saving iterations. The second one is the stopping criterion in step 3.b. The classic pegging method of Bitran and Hax [14] stops the iteration if

$$l_j \leq x_j \leq u_j \ for\ j \in I \qquad \text{or} \qquad I = \emptyset$$

However, the second stopping criterion is redundant to the first criterion because the first one covers the second one. Moreover, stopping criterion in step 3.b covers the two criteria of Bitran and Hax [14].

Remark 2.3. It is true that pegging method actually starts at a specific λ in (2.12) with full index set $I = \{1, 2, ..., n\}$. So Cominetti et al. [25] conclude that pegging method has a disadvantage because it cannot adopt an initial given λ, which may make critical improvements when (P) is used for a subproblem as proved in the experiments of [25, 31, 44]. However, it is possible to exploit pegging method with any initial λ if we peg variables by (2.14) with the initial λ before we use pegging method.

Remark 2.4. A real drawback of pegging method is that it can solve only the strictly convex problems because a non-strictly convex objective function does not have a unique center to project onto a hyperplane. This will be considered carefully in Section 4.3.1 when we present suggested method for the indefinite case of (P).

Newton method

The Newton method is quite lately developed than other methods, although it is also intuitively considerable if we draw $g(\lambda)$ as in Figure 4.5a. Maybe the reason is the well known fact that Newton method does not guarantee convergence as a root finding method. In addition to the (a) convergence issue, newton method have more serious issues: $g(\lambda)$ is (b) non-differentiable at breakpoints and (c) possibly has zero slope because $g(\lambda)$ is non-increasing. Moreover, (d) an updated λ may be placed out of the feasible bound of $[\lambda_{min}, \lambda_{max}]$. For these reasons, Dai and Fletcher (2006, [31]) did not implement Newton method although they noticed the method; instead they developed secant method.

It seems Nielsen and Zenios (1992, [88]) is the first reference which considered Newton method officially for (P), and in the same year, Robinson et al. [111] also considered it thanks to a referee's suggestion to compare with their pegging method. Lotito (2006, [74]) considered Newton method when extending the method of [88] to the semi definite case, but his work is limited by one sided bounds ($\mathbf{0} \leq \mathbf{x}$). Recently, Cominetti et al. (2012, [25]) considered Newton method in depth and utilized secant method for safeguards of the convergence issues.

Safeguards

The line of $g(\lambda)$ is not appropriate for the case of Newton method because it is not differentiable at breakpoints and can cycle or have *zero* slope. Thus, some safeguards are suggested by prior studies, summarized in Table 2.2.

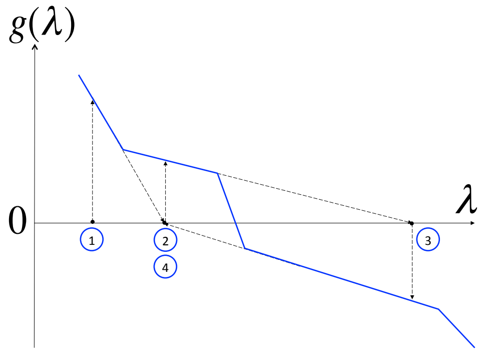
Table 2.2.: Safeguards of Newton method

	Robinson et al. [111]	Nielsen et al. [88]	Lotito [74] ($l \leq \mathbf{x}$)	Cominetti et al. [25] ($l \leq \mathbf{x}$)	Cominetti et al. [25]	Proposed method
Cycle, Out of Range	Bisection	Wrong proof	Not mentioned	Proved not happen	Secant	Secant
Non-diff-erentiable		Directional derivative by sign of $g(\lambda)$				Act like it is not
Zero slope		Not mentioned	Not mentioned	Closest left or right break point by sign of $g(\lambda)$		Secant

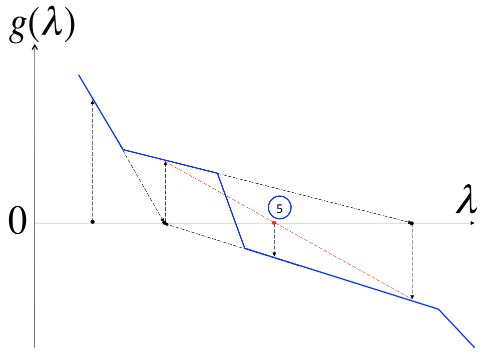
As shown in Figure 2.3a, cycles can happen only when the updated λ is not strictly inside of the updated range $[\lambda_l, \lambda_u]$. Thus, if we keep λ being updated strictly within the last range, we can resolve the cycle and out of bound issues. As a resolution, Cominetti et al. [25] suggest secant method of [31] (see Figure 2.3b) and Robinson et al. [111] choose bisection method as a safeguard. Lotito [74] does not consider the issue, but Cominetti et al. [25] proves that cycling does not happen if variables are bounded by one side such as $\mathbf{l} \leq \mathbf{x}$ or $\mathbf{x} \leq \mathbf{u}$. Nielsen and Zenios ([88]) argues that the cycle does not happen via proposition 5 in their paper, but it is wrong if their Proposition 4 is correct. Moreover, Cominetti et al. [25] show a cycle example.

The non-differentiability issue has been simply resolved by taking right derivative if $g(\lambda) > 0$ and left derivative otherwise as most literature [88, 74, 25] do; however, we will show a way to remove this issue in Algorithm 2.8.

If $g(\lambda)$ has zero slope at a λ, Newton method cannot update λ. Although [88, 74] could not notice this issue, Robinson et al. [111] suggested the convergence guaranteed bisection



(a) Newton method can cycle



(b) Secant method is used as a safeguard

Figure 2.3.: Newton method

method, and Cominetti et al. [25] proposed to use the closest right breakpoint from the current λ if $g(\lambda) > 0$ and the closest left breakpoint otherwise. However, finding the closest breakpoint takes up to $4n$ operations because it has to look through $2n$ breakpoints and compare the closeness. The cost of $4n$ is exactly the same as testing a λ point in $x_j(\lambda)$ and $g(\lambda)$ as

$$x_j(\lambda) = median\{l_j, u_j, -\lambda/d_j\} \quad \rightarrow \quad 3n\ operations$$

$$g(\lambda) = \sum_{\forall j} x_j(\lambda) - b \quad \rightarrow \quad n\ operations$$

in our transformed problem, and the cost is higher if (P) is not transformed. Therefore, as suggested by Robinson et al. [111] testing the bisection point is more efficient than taking the closest breakpoint in the sense of reducing the feasible domain of λ.

Algorithm

In this manner, it seems Newton method is not fully studied yet even though all literature concluded that the method performs well and is stable. Thus, we propose an improved method with more robust and computationally efficient safeguards in Algorithm 2.8.

For the issues of cycling and out of bounds, we suggest to utilize secant method of [31] as Cominetti et al [25] does. We can use pegging method or any convergence guaranteed root

Algorithm 2.8 $O(n^2)$; Newton method

```
1. Set initial $\lambda_l = -\infty$, $\lambda_u = \infty$, and $\lambda = -b/\sum_{\forall j}(1/d_j)$
Iterate until step 3 is satisfied
2. Get $\mathbf{x}(\lambda)$ and $g(\lambda)$
3. If $|g(\lambda)| < \varepsilon_{fea}$, finish algorithm with $\lambda^* = \lambda$ and $\mathbf{x}^* = \mathbf{x}(\lambda^*)$
4. Get $I = \{i : l_i < x(\lambda) < u_i\}$
5. If $I = \emptyset$, (slope is zero)
           Do bracketing if necessary and do secant (Safeguard 1)
   Else (slope is not zero)
     a) If $g(\lambda) > 0$, set $g_u = g(\lambda)$ and $\lambda_l = \lambda$
        Else set $g_l = g(\lambda)$ and $\lambda_u = \lambda$
     b) Update $\lambda = \lambda + g(\lambda)/\sum_{j \in I}(1/d_j)$ by Newton method
     c) If $\lambda \notin (\lambda_l, \lambda_u)$, (out of bound or possible cycle)
           Do bracketing if necessary and do secant (Safeguard 2)
```

finding methods such as bisection, sorting, or median search methods, but secant method can take the feature of the piecewise linearity of $g(\lambda)$ and our experiment shows that secant method is superior to other methods. We name it safeguard 2 in step 5.c.

To overcome the case of zero slope, we use safeguard 1 in step 5. It happens when $x_j(\lambda)$ for all j are placed at an extreme point l_j or u_j. So we can verify whether the slope is zero by looking at an index set I

$$I = \{j : l_j < x_j(\lambda) < u_j\} \tag{2.16}$$

of step 4. Cominetti et al. [25] use the closest left or right break point. It may give a good λ update, but as we mentioned earlier it takes up to $4n$ operations to update λ without the guarantee of a considerable reduction of the feasible domain, while the bisection and secant methods of [31] reduce it by half and at most 3/4 respectively. We prefer secant method to bisection method although secant method may reduce the feasible domain by less than half because it gives more reliable solutions and tends to have less iterations from experiments than bisection method.

To resolve the non-differentiability of $g(\lambda)$ at a breakpoint, all previous algorithms use the left derivative to update λ with newton method if $g(\lambda) < 0$, and the right derivative if $g(\lambda) > 0$. Two directional derivatives in (2.17) can be obtained with I in (2.16) and two index sets: $J_R = \{j : \lambda = -d_j u_j\}$ and $J_L = \{j : \lambda = -d_j l_j\}$.

$$\begin{aligned} Right\ derivative &= -\textstyle\sum_{j\in I} 1/d_j - \sum_{\lambda=-d_j u_j} 1/d_j \quad if\ g(\lambda) > 0 \\ Left\ derivative &= -\textstyle\sum_{j\in I} 1/d_j - \sum_{\lambda=-d_j l_j} 1/d_j \quad if\ g(\lambda) < 0 \end{aligned} \tag{2.17}$$

However, the probability that J_R and J_L are used is very small in practice because λ and breakpoints are continuous numbers and the possibility that an updated λ is exactly same as a breakpoint is almost practically impossible. Thus, consuming n operations at each iteration to get J_R or J_L comparing λ and n breakpoints is wasteful. For this reason, our simple resolution is acting like we never reached a breakpoint even when the current λ is at a breakpoint. So our proposed derivative is

$$Proposed\ derivative = -\sum_{j\in I} 1/d_j \tag{2.18}$$

without J_R and J_L. This may sometimes result in a worse λ update than right or left derivative; however (2.18) is more efficient if we take account of $2n$ operations to get J_R and J_L, which are highly possibly empty. Therefore, our simple proposed derivative (2.18) not only saves $2n \times$ *total iterations* operations but also makes Newton method more robust eliminating the issue of non-differentiability.

In addition to the operation saving, our safeguards also save operations to get breakpoint because our Newton method Algorithm 2.8 never uses them. Therefore, in contrast to Cominetti et al. [25], our Algorithm 2.8 can save $4n \times$ *zero slope instances* operations in safeguard 2 from getting the left or right most closest breakpoint when zero slope occurs, at least $2n \times$ *total iterations* operations from proposed derivative (2.18), and $2n$ operations to get breakpoints as summarized in Table 2.3. These savings are counted based on our transformed (P), which leads to simpler computation than other formats, so the saved operations increase if (P) is not in our transformed format.

Table 2.3.: Minimum saved operations in Newton method

	Minimum saved operations
Zero slope	$4n \times$ *zero slope iterations*
Proposed derivative	$2n \times$ *total iterations*
No need of Breakpoints	$2n$

As noticed by [25, 111] and proved by our experiment (Table 4.7), instances that need safeguards have not occurred frequently in most test problems. So the chance to save $4n \times$ *zero slope instances* operations to get the closest breakpoint as [25] does may be low. Nevertheless, our Algorithm 2.8 can save at least $2n + 2n \times$ *total iterations* operations due to a different way to get a slope.

The complexity of Newton method is $O(n^2)$ because the possible maximum iteration is $4n + 1$ (see [25] for details). In addition to the large possible iteration, Newton method also requires much more operations to update λ than other methods because it requires to update I in step 4 and compute derivatives in step 5.b.

Although Newton method may have fast convergence, because of the high operational cost in each iteration, performance of the method is sensitive to the number of iterations, and the only one parameter that we can affect is the initial λ. We suggest to use $\lambda = -b / \sum_{\forall j}(1/d_j)$ for the initial λ as step 1. It is the same value used in pegging method in (2.12), and [25] also uses it hoping that $\mathbf{x}^*$ is inside the box constraint $(\mathbf{l} \leq \mathbf{x} \leq \mathbf{u})$.

Remark 2.5. Newton method and pegging method are actually highly related with respect to the use of index sets I in (2.16) to get derivatives in Newton method and I for unpegged variables to project in pegging method. Because two index sets are essentially identical in some conditions, Cominetti et al. [25] proved that Newton method takes exactly the same iterations as pegging method if (P) has one sided bound and and $a_j > 0 \ \forall j$.

Interval test method

Interval test method is the most undeveloped method among all methods. Vavasis (1992, [132]) mentioned the interval test idea very briefly to solve the convex part of the indefinite case of (P), and Bayón et al. [7, 8] used the idea to get analytical solution on right hand side b. However, no one has clearly developed an efficient implementation and compared with other methods. Thus, we clearly present an efficient procedure of interval test method in Algorithm 2.9.

Optimum conditions

The main difference between interval test method and other Lagrange multiplier search methods is the way to reduce the domain of Lagrange multiplier. Other multiplier search methods reduce the dual domain by a point substituting a λ and update it by the sign of $g(\lambda)$ until it gets a feasible solution. On the other hand, interval test method uses breakpoints to construct intervals on the line of λ and reduces the domain of Lagrange multiplier by a sub domain (interval or range) in each iteration. This is possible because (P) is a separable knapsack problem and $x_j(\lambda)$ has a unique value at a given λ. Suppose that breakpoints are sorted in ascending order and we have an interval $[\hat{\lambda}_l, \hat{\lambda}_u]$ that consists of two consecutive breakpoints as

$$[\hat{\lambda}_l, \hat{\lambda}_u] \subseteq [\lambda_l, \lambda_u]$$

Then, the index set $I = \{j \in P : l_j < x_j(\bar{\lambda}) < u_j\}$ is unchanged for a $\bar{\lambda}$ that is strictly within the interval $(\hat{\lambda}_l, \hat{\lambda}_u)$ because $x_j(\lambda)$ is a decreasing function between its two breakpoints or constant at l_j or u_j. With the index set I, we can define $g(\lambda)$ as

$$\begin{aligned} g(\lambda) &= \sum_{j\in L} l_j + \sum_{j\in U} u_j - b - \lambda \sum_{j\in I}(1/d_j) \\ &= H - \lambda G \end{aligned} \tag{2.19}$$

where

$$\begin{aligned} L &= \{j : x_j(\bar{\lambda}) = l_j\} & U &= \{j : x_j(\bar{\lambda}) = u_j\} \\ H &= \sum_{j \in L} l_j + \sum_{j \in U} u_j - b & G &= \textstyle\sum_{j \in I}(1/d_j) \end{aligned} \tag{2.20}$$

Moreover, if $\lambda^* \in [\hat{\lambda}_l, \hat{\lambda}_u]$, then

$$g(\lambda^*) = 0 = H - \lambda^* G$$

, and it implies

$$\lambda^* = H/G$$

This is proved by Lemma 2.6 and Theorem 2.7 and applied in step 5.a and step 5.c of Algorithm 2.9.

Lemma 2.6. *Define an interval $[\hat{\lambda}_l, \hat{\lambda}_u]$ which consists of two adjacent breakpoints and $\hat{\lambda} = H/G$. Then,*

$$\hat{\lambda}_u < \lambda^* \quad \textit{if and only if} \quad \hat{\lambda}_u < \hat{\lambda} \tag{2.21}$$

$$\lambda^* < \hat{\lambda}_l \quad \textit{if and only if} \quad \hat{\lambda} < \hat{\lambda}_l \tag{2.22}$$

Proof. If $g(\lambda) > 0$ for a $\lambda \in [\hat{\lambda}_l, \hat{\lambda}_u]$, then $\hat{\lambda}_u < \lambda^*$ and $g(\hat{\lambda}_u) > 0$ since $g(\lambda)$ is a non-increasing function. This results in

$$\begin{aligned} 0 &< g(\hat{\lambda}_u) = H - \hat{\lambda}_u G \quad \textit{since } G > 0 \\ \hat{\lambda}_u &< H/G = \hat{\lambda} \end{aligned}$$

Similarly, if $g(\lambda) < 0$ for a $\lambda \in [\hat{\lambda}_l, \hat{\lambda}_u]$, then $\lambda^* < \hat{\lambda}_l$ and $g(\hat{\lambda}_l) < 0$ since $g(\lambda)$ is a non-increasing function. This results in

$$\begin{aligned} 0 &> g(\hat{\lambda}_l) = H - \hat{\lambda}_l G \quad \textit{since } G > 0 \\ \hat{\lambda}_l &> H/G = \hat{\lambda} \end{aligned}$$

□

Theorem 2.7. *Let $\hat{\lambda} = H/G$. If $\hat{\lambda} \in [\hat{\lambda}_l, \hat{\lambda}_u]$, then $\lambda^* = \hat{\lambda}$.*

Proof. If we have an interval $[\hat{\lambda}_l, \hat{\lambda}_u]$ such that

$$g(\hat{\lambda}_u) \leq 0 \leq g(\hat{\lambda}_l)$$

, then by Lemma 2.6 it holds that

$$\hat{\lambda}_l \leq H/G \leq \hat{\lambda}_u$$

Therefore, the theorem is proved. □

Algorithm and its efficiency

To construct intervals, we first select breakpoints that are within $[\lambda_l, \lambda_u]$, and then sort the selected breakpoints as in step 4 of Algorithm 2.9. The sorting procedure dominates the complexity of interval test method by $O(n\ log\ n)$ as the main disadvantage. However, interval test method can be practically efficient if the number of breakpoints to be sorted is small. It is affected by two factors: (a) an initial $[\lambda_l, \lambda_u]$ and (b) the distribution of breakpoints around λ^*. The distribution is uncontrollable because it is determined by the given coefficients of problem, but we can control the initial range. The initial range can be $[\lambda_{min}, \lambda_{max}]$ or obtained by any root finding methods that we have considered, but secant method would be the best choice because it guarantees to find a bracket that contains λ^* efficiently, and our experimental results show that it has the fast convergence. Newton method also tends to have fast convergence, but we prefer secant method because Newton method requires much more operations to update λ and uses secant method as a safeguard. The superior performance of use of secant method for the bracket is proved by experiments.

Once we construct intervals within a range of $[\lambda_l, \lambda_u]$, we have values of $x_j(\lambda)$ and $g(\lambda)$ on

Algorithm 2.9 $O(n\ log\ n)$; Interval test method

```
1. Obtain $[\lambda_l, \lambda_u]$ $(\lambda_u - \lambda_l < \varepsilon_{bra})$, $\lambda$ $(= \lambda_l$ or $\lambda_u)$, $x(\lambda)$, and $g(\lambda)$ using methods such as
   secant method.
2. Initialize $H = 0$ and $G = 0$, and get breakpoints $T = \{-d_j u_j, -d_j l_j\}$
3. Get $I = \{j \in P : l_j < x_j(\lambda) < u_j\}$, $H = \sum_{j \notin I} x_j(\lambda) - b$, and $G = \sum_{j \in I}(1/d_j)$
4. Get $R = \{j : \lambda_l \le T_j \le \lambda_u\}$, $[T, R] = sort(T_j : j \in R)$ in ascending order
   where $T$ is sorted $T_j$ for $j \in R$ and $R$ is corresponding index set of sorted $T$.
5. If $g(\lambda) \ge 0$, iterate for $i = 1$ to $|T|$, set $j = R_i$
     a) If $G \neq 0$ (nonzero slope), set $\hat{\lambda} = H/G$
          i. If $\hat{\lambda} \le T_i$, finish algorithm with $\lambda^* = \hat{\lambda}$ and $\mathbf{x}^* = \mathbf{x}(\lambda^*)$
     b) If $j \le n$ (if $T_i = -d_j u_j$), update $H = H - u_j$ and $G = G + 1/d_j$
        Else (if $T_i = -d_j l_j$), update $j = j - n$, $x_j^* = l_j$, $H = H + l_j$ and $G = G - 1/d_j$
   Else $(g(\lambda) < 0)$, iterate for $i = |T|$ to 1, set $j = R_i$
     c) If $G \neq 0$ (nonzero slope), set $\hat{\lambda} = H/G$
          i. If $\hat{\lambda} \ge T_i$, finish algorithm with $\lambda^* = \hat{\lambda}$ and $\mathbf{x}^* = \mathbf{x}(\lambda^*)$
     d) If $j \le n$ (if $T_i = -d_j u_j$), update $x_j^* = u_j$, $H = H + u_j$ and $G = G - 1/d_j$
        Else (if $T_i = -d_j l_j$), update $j = j - n$, $H = H - l_j$ and $G = G + 1/d_j$
6. (at leftmost or rightmost interval) Finish with $\lambda^* = H/G$ and $\mathbf{x}^* = \mathbf{x}(\lambda^*)$
```

hand with $\lambda = \lambda_l$ or λ_u. So we are already on the leftmost interval if $g(\lambda) \ge 0$ or rightmost interval if $g(\lambda) \le 0$ on λ. At the current interval, we calculate $\hat{\lambda} = H/G$ if $G \neq 0$ and test if $\hat{\lambda}$ is within the current interval as step 5.a and step 5.c. If it is within the interval, it is the optimum λ^* thanks to Theorem 2.7. If it is not, we need to update only H and G. For example, if we are on an interval $[\hat{\lambda}_l, \hat{\lambda}_u]$ where is strictly left from λ^* (that is, $\hat{\lambda}_u < \lambda^*$), we first check where $\hat{\lambda}_u$ is from. If it is the value of $-d_j u_j$, then the index j should be switched from a member of U to I. That implies to update H and G by

$$H = H - u_j \text{ and } G = G + 1/d_j$$

, and all four cases are presented in step 5.b and step 5.d.

The reason why we decide the first interval by the sign of $g(\lambda)$ in step 1 is the efficiency. When the last $g(\lambda)$ is obtained by either the last λ_l or λ_u, $\mathbf{x}(\lambda)$ is also obtained. Then, the optimum solution $\mathbf{x}^*$ can be acquired easily switching one variable from either l_j, u_j or (l_j, u_j) to another while H and G are updated in step 5.b and step 5.d. This is consistent to the fixing Algorithm 4.5 that will be considered in Section 4.3.1 as a convergence accelerator.

The update procedure makes the interval test method extremely efficient since it requires to updates only H and G to find $\hat{\lambda}$ and decide the optimality with only 5 or 6 operations in our experiments. This is a big advantage comparing with other methods. See the Table 2.4. It shows the worst case number of operations in the iteration part of all considered methods except for interior point method and an example of total operations with $n = 100$ and moderate values (see the Table description for values). It shows that interval test method requires the extremely smaller operations than other methods (say, about 3 to 224 times smaller operations) in the worst case. Therefore, although interval test method requires to sort a part of breakpoints, it can find the optimum solution very efficiently once it construct intervals via a methods such as secant method.

Remark 2.8. We can see that $-G$ is actually the slope of the piecewise liner function $g(\lambda)$ strictly within a testing interval $(\hat{\lambda}_l, \hat{\lambda}_u)$ as same as the proposed derivative (2.18) in Newton method. Thus, if $G = 0$ in step 5.a and step 5.c of Algorithm 2.9, the slope of $g(\lambda)$ is *zero* and the testing interval cannot contain $\lambda^* \notin (\hat{\lambda}_l, \hat{\lambda}_u)$. So we just skip the interval updating H and G for the next interval.

Remark 2.9. We have considered jump interval test method that tests intervals jumping the computationally optimum number of intervals forward and backward until it converges to save computations to get $\hat{\lambda}$ and check its optimality in step 5.a and step 5.c of Algorithm 2.9. The idea helps interval test method perform little better; however, we do not present it because it is proved that the performance improvement is not always positive and practically negligible through our experiments. So we show a simpler and competitive version of interval test methods in Algorithm 2.9.

Table 2.4.: The worst case number of operations in iteration parts

	Maximum Iteration	Computation per Iteration	Total in n	Total with $n = 100$
Interval test[1]	$2n$	5.5 (5 for n times and 6 for n times)	$11n$	1100
Sorting[2]	$\lfloor 1 + log_2(2n) \rfloor$	$3 + 4n$		3224
Median search[2]	$\lfloor 1 + log_2(2n) \rfloor$	$1 + 4n$ and Median search		3208 + Median search
Secant[3]	Not measurable	$16 + 4n$ or $9 + 4n$		6240
Bisection[4]	$\left\lceil log_{0.5}\left(\frac{\varepsilon_{gap}}{\lambda_{max}-\lambda_{min}}\right)\right\rceil$	$3 + 4n$		12090
Pegging[5]	n	$3 + 8\hat{n}$ ($\hat{n}$: the number of unpegged var.)	$4n^2 + 7n$	40700
Newton[6]	$4n + 1$	$3 + 6n$ and fixed $n(n+1)/2$	$24.5n^2 + 18.5n + 3$	246853

1. There are at most $2n + 1$ intervals in interval test method so at most $2n$ iteration is required.
2. Iteration of sorting and median search method is $8 = \lfloor 1 + log_2(2n) \rfloor$ with $n = 100$
3. Secant method takes usually much less than 15 iterations for the size of $n = 2e6$ from literature and our experiments. So 15 iteration may be enough to assume the worst case for $n = 100$.
4. With $\varepsilon_{gap} = 1e - 7$ and $\lambda_{max} - \lambda_{min} = 100$, bisection method has at most $30 = \left\lceil log_{0.5}\left(\frac{\varepsilon_{gap}}{\lambda_{max}-\lambda_{min}}\right)\right\rceil$ iterations.
5. Pegging method decreases $\hat{n}$, the number of index for unpegged variables, by one every iteration in the worst case. If we assume that $1/d_j\ \forall j$ is computed before iteration, the number of operations is computed by $3n + 8\left(\sum_{i=1}^{n} i\right) = 3n + 8n(n+1)/2 = 4n^2 + 7n$. This is acquired based on Kiwiel [65] and our implementation in the code, whichever is computationally more efficient.
6. In Newton method, if we assume that $1/d_j\ \forall j$ is computed before iteration and only one variable is excluded from I in each iteration, then it needs at most $n(n+1)/2$ $(= \sum_{i=1}^{n} i)$ computations to get our proposed derivative $\sum_{j \in I}(1/d_j)$ in (2.18). So the total number of operations is $(1 + 4n)(3 + 6n) + n(n+1)/2 = 24.5n^2 + 18.5n + 3$.

Interior point method

Wright and Rohal (2013, [136]) developed an interior point method for a problem that consists of twice differentiable separable convex objective function and a knapsack and box constraints so it covers the problem (P). Their algorithm basically solves KKT system applying primal-dual interior point method searching for directions and step sizes with barrier parameters. They argues that "the key to making it competitive is the fact that the direction search (efficient linear-system solution in the Newton update) can be found in Cn arithmetic operations for a small fixed value of C", and their experiment results show that interior point method outperforms the median search method.

However, the purpose of the algorithm is to solve a separable general resource allocation problem that does not have a closed-form solutions like $\mathbf{x}(\lambda)$ so its subproblem must be solved numerically. Thus, a quadratic problem (P) is not considered among 10 problem classes in their experiments. Above all, they left the proof for convergence and complexity for the future research. For all these reasons, we do not consider interior point method as a competitor to solve (P) in this chapter.

2.2.3. Geometric interpretation

As discussed in Chapter 1, the solution of the strictly convex case of (P) is the closest point from the ellipsoidal center of the objective function to the feasible domain which is intersection of the hyperplane (knapsack constraint) and box constraints. This section gives geometric interpretations for Lagrange multiplier search methods, pegging method, and interval test method with an example. This interpretation is worth better understanding the difference of methods. Consider an example problem

$$
\begin{aligned}
(P_{ex}) \quad Min \quad & \frac{1}{2}\mathbf{x}'\mathbf{x} \\
s.t. \quad & x_1 - 2x_2 = 0.8 \\
& \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}
\end{aligned}
$$

The contours of objective function of the strictly convex case of (P) are ellipsoids as we stated in the introduction, but in the case of (P_{ex}), the contours are circles centered at the origin $(0, 0)$ because its $\mathbf{D}$ is an identity matrix in two dimensional domain. Then, we can geometrically interpret (P_{ex}) in two ways:

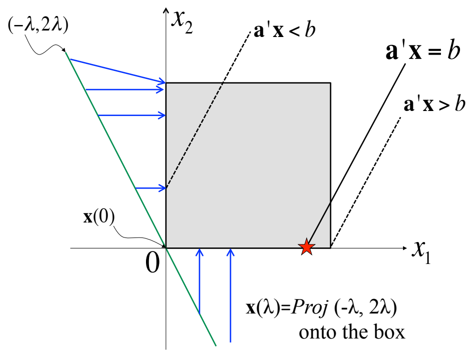
1. Find a smallest circle (hypersphere) that touches the intersection of the hyperplane and box constraint.

2. Find a closest point from the center (the origin in this case) to the intersection of the

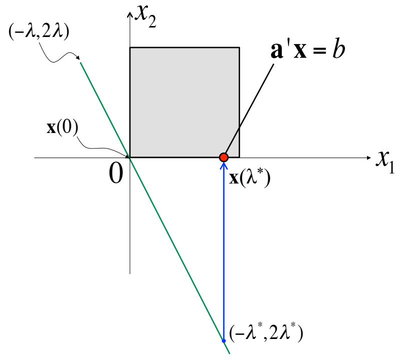
hyperplane and box constraint.

Lagrange multiplier search method

The example problem (P_{ex}) can be drawn on two dimension as Figure 2.4a. The box constraint $\mathbf{0} \leq \mathbf{x} \leq \mathbf{1}$ and the knapsack constraint $\mathbf{a}^T\mathbf{x} = b$ appears as a shade box and a line. So the feasible domain is the line in the box. If we draw a circle centered at the origin $\mathbf{0}$, the smallest circle touches the optimum solution point dotted by a star on the feasible area.



(a) Stay on $(x_1, x_2) = (-\lambda, 2\lambda)$



(b) Project $(-\lambda, 2\lambda)$ onto the box

Figure 2.4.: Lagrange multiplier search method

Following $\mathbf{x}(\lambda)$ in (2.6), the solution of (P_{ex}) can be determined by

$$\begin{aligned} x_1 &= median\{0, 1, -\lambda\} \\ x_2 &= median\{0, 1, 2\lambda\} \end{aligned} \tag{2.23}$$

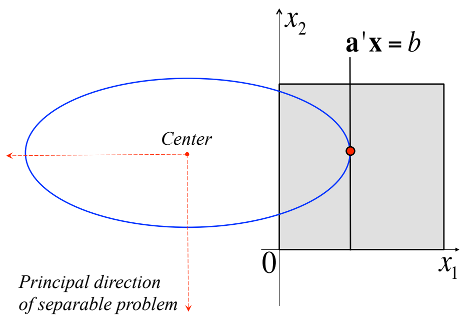
This means we specify a point on the line (plane) of $(x_1, x_2) = (-\lambda, 2\lambda)$ with a given λ as Figure 2.4a, and then the point is projected onto the box by (2.23). Thus, Lagrange multiplier search methods basically tries a λ and projects it onto the box until it meets the optimal solution, which satisfies knapsack constraint as Figure 2.4b.

This simple projection function $\mathbf{x}(\lambda)$ works because the principal direction of the ellipsoid of objective function $\mathbf{x}^T\mathbf{D}\mathbf{x}$ is same as the box constraint. A principal direction of ellipsoid can be obtained by eigen vectors. The orthonormal eigen vectors of the diagonal matrix $\mathbf{D}$ is always the identity matrix $\mathbf{I}$, and eigenvalues d_j give the weight to form an ellipsoid. Since the box constraint can be expressed by

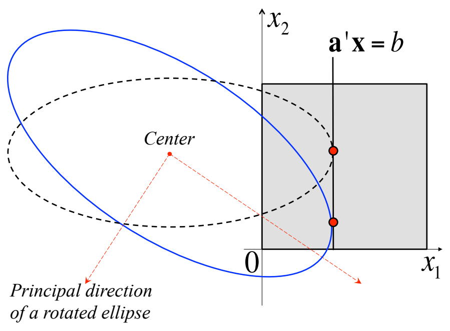
$$\mathbf{l} \leq \mathbf{I}\mathbf{x} \leq \mathbf{u}$$

, the principal directions of the ellipsoid and the box constraints are same as $\mathbf{I}$.

If we think of the non-diagonal matrix $\mathbf{D}$, its eigen vectors are no longer the identity matrix. Thus, the ellipsoid is rotated or flipped by its principal direction, and a projected point from a point to box is not a point on the smallest ellipsoid since its size is changed on its principal direction. This is the geometric reason why we cannot get a simple projection function $\mathbf{x}(\lambda)$ in the case of non diagonal matrix $\mathbf{D}$. We call the case the nonseparable quadratic programming. See an example on Figure 2.5.



(a) Separable case



(b) Nonseparable case with rotated ellipse

Figure 2.5.: Separable and Non-separable case

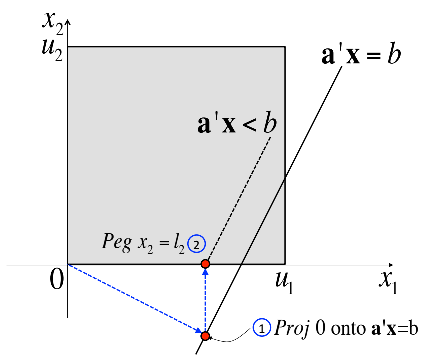
Two ellipses on Figure 2.5 have the same center but different principal directions. Since the principal directions of the ellipse in Figure 2.5a are unit vectors, the smallest ellipse that touches the feasible domain is the projected point from the center to the feasible domain; however, since the ellipse in Figure 2.5b is rotated with non-unit vector principal direction, the point that the smallest ellipse touches the feasible domain is not the projection of center.

Therefore, a simple projection function $\mathbf{x}(\lambda)$ could not exist in the nonseparable quadratic programming.

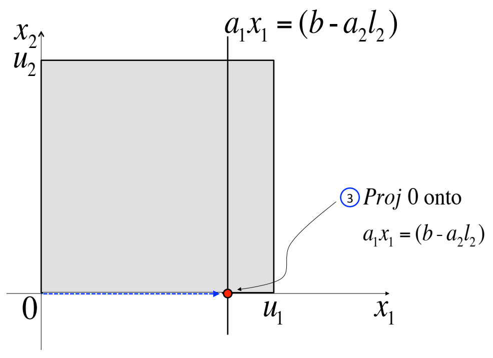
Pegging method

Lagrange multiplier search method projects a point $(x_1, x_2) = (-\lambda, 2\lambda)$ onto the box, while pegging method first projects the center of the objective function onto the knapsack constraint and then project the point again onto the box. If the box projected point violates the knapsack constraint, variables are pegged depending on the sign of violation as Figure 2.6a.

The two projection steps and pegging are iteratively done as Figure 2.6b until all unpegged


x_2
u_2
$\mathbf{a}'\mathbf{x} = b$
$\mathbf{a}'\mathbf{x} < b$
Peg $x_2 = l_2$ ②
0
u_1
x_1
① *Proj* 0 onto $\mathbf{a}'\mathbf{x}$=b

(a) Project center onto knapsack constraint and peg a variable



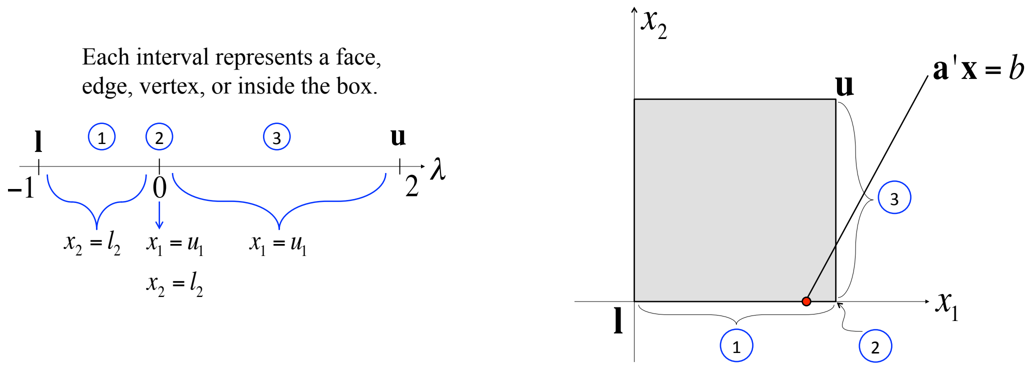
(b) Project center onto knapsack constraint with unpegged variables

Figure 2.6.: Pegging method

variables are projected within the box. Another similar geometric interpretation is nicely presented by Robinson et al. [111], and they call the method a series of restricted projection problems.

Since pegging method first projects its center onto the knapsack constraint, it must have a unique center. That is why all literature about pegging method for non-quadratic objective functions also assume its objective function is strictly convex and twice differentiable. For the same reason, if (P) is in positive semidefinite case, pegging method is no longer capable to solve (P) because the center of the objective function is not unique, while Lagrange multiplier search methods can find a $\mathbf{x}(\lambda)$ and all global optimum solutions $\mathbf{x}^{U*} = \{\mathbf{x} \in X_U : \mathbf{a}^T\mathbf{x} = b\}$ by the global case 1 of Section 4.2.3.3.

Interval test method



(a) Construct intervals (b) Travel on the box

Figure 2.7.: Interval test method

While the previous two methods use projection, interval test method does not use it. Interval test method first constructs intervals on λ using breakpoints. Breakpoints of (P_{ex}) are $\{-1, 0, 0, 2\}$ and intervals can be constructed as Figure 2.7a. Since $\lambda^* \in [-1, 2]$, each interval is a subspace of dual feasible domain. At an interval, we can get three index sets I, L, and U, which are used in (2.19) for interval test method. If we use L and U to fix x_j to l_j or u_j and I to allow x_j to be placed within $[l_j, u_j]$, all $\mathbf{x}$ points stay on a face of the box. Therefore, an interval represents a face, vertex, edge, or inside of the box, and breakpoints indicate a vertex of the box since $I = \emptyset$ at a breakpoint as Figure 2.7b.

Moving form an interval to another interval actually means we travels a face to a connected face of the hyperbox looking for an optimal solution. For example, if we are on the leftmost interval ① in Figure 2.7a, then we investigate whether an optimal solution exists on the face ① in Figure 2.7b. Although the number of faces of a hyperbox[4] is $n(n-1)2^{n-3}$, we travels much smaller number of faces since at most distinct $2n$ break points generate only at most $2n - 1$ intervals.

[4]A hyperbox has $n2^{n-1}$ edges and 2^n vertices.

2.2.4. Which one is the best?

We have considered nine methods to solve the strictly convex case of (P). If we count only complexity, $O(n^2)$ time pegging and Newton method should have the worst performance theoretically. However, it is not always true in practice. According to the experiment results of Cominetti et al. [25], theoretically leanest $O(n)$ time median search method is always slower than other theoretically most expensive pegging and Newton methods, and Experiment results of Kiwiel [65] shows that pegging method is up to 1.14 times faster than his median search method in [63].

This result is also supported by survey of Bretthauer and Shetty (2002, [16]) and Patriksson (2008, [106]). Both survey literature concluded that pegging method is generally performs better than Lagrange multiplier search methods if a problem has a closed form like $\mathbf{x}(\lambda)$.

Because experiment result is highly affected by environment such as test problems, computer language, coding style, and computer system, we need to test multiple methods in the same environment to compare performances. Comprehensive experiments have been done by Robinson et al. (1992, [111]), Kiwiel (2007, [63], 2008, [64, 65]), and Cominetti et al. (2012, [25]). The results are summarized in Table 2.5, and it also includes results of literature that compared more than two methods. According to the Table, it seems pegging is prior since most experiments concluded it is faster than others. More detailed experiment results are described from the recent paper below.

Note that a recently developed interior point method of Wright et al. (2013, [136]) is not included in Table 2.5 although they concluded that their method outperforms medians search method because they did not tested the quadratic programming.

2012 Cominetti et al. [25] compared their Newton method with all recent algorithms: the secant method (2006, [31]), median search method (2008, [64]), and pegging method (2008, [65]). Bisection, Sorting, and Approximate median search methods may be excluded because it has been known that those methods are practically and theoretically

Table 2.5.: History of experiment for strictly convex case

Year [Cite]	Bisect	Sorting (Rank)	Exact Median	Approx. Median	Secant	Newton	Pegging (Var. fix.)
2012 [25]			[64]		[31]	⋆, ≈	⋆, ≈ [65]
2008 [64]			⋆ own [20, 21]	own			
2008 [65]			≈ [63]				⋆
2007 [63]			⋆ own [20, 21]				
2006 [31]			[101][3]		⋆		
2006 [74]			[20]	[101]		⋆	
2002 [17]		[18]					⋆
1998 [137]	≈ [117]				≈ [117]	≈ [117]	
1997 [15]		[50]					⋆
1996 [19]		[50]					⋆
1992 [111]		[50]	[20]	[101]		≈	⋆
1992 [88]		[50]				⋆	
1991 [134]		[50]		own			⋆ [14]
1990 [101]		[50]		⋆			
1989 [133]	[27]						⋆ [14]

1. Citation in each cell is the algorithm that was used for the corresponding paper's experiments.
2. In the corresponding paper's experiments, ⋆ means the fastest method; ≈ means comparable method to the fastest one; "own" means tester's own method
3. Dai and Fletcher [31] used exact median search based on the procedure in the approximate median search method in Pardalos and Kovoor [101].

performs worse than others. Because the experiments have been done for all recent algorithms with recently considered problems in [31, 64, 65], we state detail of the experiment results. Cominetti et al. [25] conducted tests with two sets of random problems and support vector machine (SVM) problems as a sub-problem.

- Two random problems
  - The random problem of [65] was used to test integer problems by Bretthauer and Shetty (1995, [18]). With the random problem, Cominetti et al. [25] reported their Newton method is about 20% faster than others. Ironically the theoretically most efficient method, exact median search method is the slowest one. "Newton method takes significantly smaller iteration than others, but the running time does not reflect it because Newton method needs to compute slopes at each

iteration" [25]. The order of speed in this random problems is Newton (fastest) - Pegging - Secant - Median search (slowest).

  - The other random problem is quite practical because it is oriented from multicomodity network flows and logistics problems, which were similarly used in [31]. The result shows that Newton method is beaten by Pegging method. It is because Newton and pegging methods have the almost same iterations and Newton method requires to compute derivative in each iteration. The order of speed in this random problem is Pegging (fastest) - Newton - Secant - Median search (slowest).

- Support vector machine (SVM) problems
  - Cominetti et al. [25] did the same experiments that [31] did with a real SVM problems of MNIST[5] and UCI Adult[6]. SVM problem is in the form of (P) but the quadratic terms are not separable. So SVM problems are solved by spectral projected gradient (SPG) method of [13], and (P) is solved multiple times as a sub-problems. The order of speed in SVM problems is Newton (fastest) - Pegging - Secant - Median search (slowest).
  - In addition, Cominetti et al. [25] also suggested a new way to give an initial λ (hot start) for the next iteration, while [31] just uses the last λ for the initial value in the next iteration. The new hot start saves CPU time about 45% with Newton method and about 57% with secacnt method.

2008-2007 Kiwiel [64] investigated the median search method in depth based on his previous research on median search algorithm SELECT [66] and improved the median search method proving that other median search methods [26, 52, 77, 76, 101] may not converges. The improvement of his median search method is proved by the experiment comparing five versions of median search methods: his own method with (1) exact median and with (2) approximate median method, (3) modified and (4) original Brucker

[5]http://yann.lecun.com/exdb/mnist/
[6]http://archive.ics.uci.edu/ml/datasets/Adult.

[20], and (5) Calamai-More [21]. The experiment shows that Kiwiel's exact median search method is the best, but we may not consider that the difference is significant because his best method is less than 1.3 times faster than other methods in the large enough size of $n = 2,000,000$. Kiwiel published a similar median search method paper (2007, [63]) one year before his paper [64] and compared with two median search methods: Brucker [20] and Calamai-More [21].

2008 Kiwiel [65] also studied the pegging method in addition to his work [63, 64] on the median search method. According to his experiment, his pegging method is slightly faster (up to 1.14 times) than his exact median search method [63] on average, but exact median search method is more stable with smaller range of CPU times. Kiwiel [65] says it is due to the high efficiency of median search routine [66]. This experiments may conclude that Kiwiel's pegging method is better than Kiwiel's medians search method on average CPU time.

2006 Dai and Fletcher [31] is the first paper which explicitly applied secant method for (P). The secant method was compared with the exact median search , which is based on Pardalos and Kovoor [101][7]. The experiments shows that secant method is up to about 5 times faster than the exact median search method.

2006 Lotito [74] tested his own Newton method following the methods of [88] versus exact median search method [20] and approximate median search method [101]. He concluded his own Newton method is faster and more robust than other methods.

2002 Bretthauer and Shetty [17] compared sorting methods of [50, 18] and pegging method in order to solve (P) with integer variables. The continuous version of (P) is used to do branch and bound for integer solutions. In the experiment results, pegging method is $3 - 4$ times faster for uncorrelated, weakly, and strongly correlated problems and $1.5 - 1.6$ times faster for manufacturing capacity planning problems .

[7] Pardalos and Kovoor [101] suggested the approximate median search method, but Dai and Fletcher [31] replaced the approximate median search with the exact median search for their experiment.

1998 Wu et al. [137] used two hybrid methods, Newton-secant and Secant-bisection of [117], to find a step size of dual ascent algorithm. Two hybrid methods use two multiplier search methods sequentially[8]. A statistical test, two-way analysis of variance, shows that two methods does not have significantly different performance.

1997 Bretthauer and Shetty [15] developed a method to solve a generalized upper bound (GUB problem), and a method is required to solve (P) as a sub-problem. So authors tested sorting methods [50] and their pegging method, which is a modified version of [111] to avoid transformation such that $d_j = 1$ and $c_j = 0$. In the test results, their pegging method is faster than sorting method [50] and approximately 4000 times faster than a general purpose nonlinear programming software LSGRG [122].

1996 Bretthauer et al. [19] suggested a modified version of a pegging method [111]. The purpose of modification is to avoid the problem transformation in [111] whenever their branch and bound algorithm solves (P) in every node. In their three correlated test problems, pegging method is $3.2 - 3.7$ times faster than sorting method.

1992 Robinson et al. [111] did comprehensive experiments with all accessible methods at that time except bisection method. The test results show that his pegging algorithm is superior to approximate median search method [101], sorting method [50], and exact median search method [20] and is comparable to Newton method. Robinson et al. [111] tested Newton method by a referee's suggestion, and it is believed that it is the first time to include Newton method in experiments.

1992 Nielsen and Zenios [88] tested four methods including Newton and sorting [50] methods and is the only one that tested Bregman projection algorithm and Tseng method [130]. The experiment results show that the performances are mixed. However, authors concluded that Newton method performs consistently well because it usually takes fewer iteration than others.

[8]For example, it solves a problem by Newton method for iteration 1, secant method for iteration 2, Newton method for iteration 3, and so on

1991 Ventura [134] compared sorting method [50] with MergeSort, approximate median search method[9], and pegging algorithm [14] to solve the multidimensional quadratic programming with (P) as a sub-problem to find the optimal step size. His experiment shows that pegging is the best and sorting method is the worst.

1990 Pardalos and Kovoor [101] developed the approximate median search method and compared with a sorting Algorithm [50]. The experiments approved their argument that the approximate median search method has expected $O(n)$ complexity although the worst case complexity is $O(n^2)$ with 500 random problems, which may be large enough to show the expected complexity. However, they did not compare with the linear time exact median search method [20] which is noticed as an $O(n)$ time method by themselves.

1989 Ventura [133] compared bisection method [27], pegging method [14], and a hybrid method. The hybrid method initially use pegging method and switch to use bisection method. The hybrid method performs better than two pure methods for all instances, and pegging method is better than bisection method. But the author could not find the good transition strategy.

2.3. Positive Semidefinite case

In this section, we consider the positive semidefinite case of (P). That is, $d_j \geq 0 \; \forall j$. As we shall show in the global case 1 of Section 4.2.3.3, we can find a set of all global optimum solutions due to the strong duality theorem.

A recent literature Zhang and Hua (2012, [140]) stated that they could not find any specialized method for this case of (P), but this case actually has been considered in a few literature [31, 74, 132] since 1992.

It seems Vavasis (1992, [132]) is the first one who considered this case. He presents $x_j(\lambda)$ for $j \in Z$ as (4.5) and suggests the interval test method to find the λ^*. Over one decade later,

[9]Ventura (1991, [134]) named this the random search, and he did not cite Pardalos and Kovoor (1990, [101]), who developed the approximate median search method at the first time in a similar moment.

Dai and Fletcher (2006, [31]) also suggest a different interval test method that construct intervals only with breakpoints from $d_j = 0$. In the same year, Lotito (2006, [74]) studied the semidefinite case with Newton method presenting λ_{min} that is narrowed by variables in Z as Lemma 4.5, but his work is limited by one sided bound constraints ($\mathbf{0} \leq \mathbf{x}$).

Zhang and Hua (2012, [140]) utilize bisection method to solve the case with inequal knapsack constraint in a continuous space and present two heuristic methods to find integer solutions based on the continuous solution. However, their contribution is quite limited because of four reasons. (a) By Lemma 4.5, we can narrower the dual domain by breakpoints from variables in Z; however, [140] finds λ_{max} from the maximum value of all breakpoints and sets λ_{min} just *zero* although breakpoints from Z can give greater value of λ_{min} than *zero*. (b) Thus, its possibly larger dual domain makes the performance of its bisection method worse resulting in more iterations. (c) Above all, bisection method is numerically unreliable to solve the non-strictly convex case because multiple distinct breakpoints t_j for $j \in Z$ can exists within ε_{gap} range of the last interval $[\lambda_l,\ \lambda_u]$. So it can easily miss the global optimum for relatively easy separable convex problem. (d) For the last, although global solutions are a subset of X_U in (4.16) that satisfies the knapsack constraint, it restricts its solution to only one. It is presented in its proposition 5, but even it is wrong. In addition to the error, its proposition 6 is also not correct because it says $g(\lambda)$ is decreasing although it is non-increasing possibly having zero slope.

Experiments have been done only by Zhang and Hua (2012, [140]) for the positive semidefinite case, and they compared their bisection method with general purpose quadratic solvers: `quadprog.m` of MATLAB and `mskqpopt.m` of MOSEK. The results says the speed of [140] is about 10 times faster, yet it does not mean that their algorithm is better than two commercial solvers because it is specially designed to solve the simple structure problem (P).

2.4. Nonconvex case

If there exist some negative d_j ($d_j < 0$ for some j), problem (P) is an *NP-hard* problem. Pardalos and Vavasis [104] proved that (P) is an *NP-hard* problem even if a single d_j is negative. Moreover, Pardalos and Schnitger [103] proved that checking local optimality in constrained quadratic programming is *NP-hard* as its paper title.

Although it is very hard to solve, many researchers have challenged to develop global solvers for the general (nonconvex, nonseparable, and multidimensional constrained) quadratic programming by recently because of its usefulness. See [102] for its enormous applications. The difficulty also results in various techniques and methods. As a recent research Chen and Burer (2012, [23]) developed a global solver combining ideas of branch-and-bound and copositive programs. A new interior point method was developed by Absil and Tits (2007, [1]). Chinchuluun et al. (2005, [24]) applied approximation set method, and Phong et al. (1995, [107]) used branch-and-bound techniques. Methods that employ benders cut method were presented by Kough (1974, [68] 1979, [69]). A good overview of important methods and applications is available in Pardalos (1991, [99]) with description about Benders decomposition, concave programming approaches, enumerative techniques, and bilinear programming. One can also refer a book of Pardalos and Rosen [102] and a survey of Pardalos and Rosen [97], and Absil and Tits [1] has a good introduction with history.

As similar to (P), the separable (nonconvex, nonlinear, and multidimensional constrained) programming has been studied by Illés and Nagi (2005, [55]) and Xue et al. (2004, [138]), and [55] listed numerous literature for three most used algorithms: branch-and-bound, Vertex enumeration, and cutting-plane methods. For the quadratic objective function and linear constraints, Rosen and Pardalos (1986, [114]) and Barrientos and Correa (2000, [5]) developed methods for separable programming to solve the general quadratic programming by diagonalizing procedure of [114].

Recent studies that focused on knapsack (nonconvex, nonlinear and nonseparable) programming are the multiple-choice knapsack programming of Sharkey (2011, [118]), the decomposition methods for large size problems of Lin et. al (2009, [72]), and the locally Lipschitz continuous objective function of Romeijn et al. (2007, [113]). The knapsack quadratic programming is researched by Nowak (1998-1999, [90, 91, 92]) for the simplex constraint, by Pardalos (1988, [98]) for the quadratic programming on a convex hull, and by Pardalos et al. (1991, [105]) for two global interior point methods.

The papers that focus on indefinite KSQP are (1990-1992 [83, 101, 131, 132]) to our best knowledge, and it is supported by a recent survey of Bretthauer and Shetty (2002, [16]) and the encyclopedia of optimization (2009, [139]).

2.4.1. Algorithms for nonconvex case

All four methods [83, 101, 131, 132] specialized for the nonconvex KSQP are introduced in this section. Briefly [83] developed a local solver for the strictly concave case, and one of its authors, Vavasis extended it to a local solver for the indefinite case in [132]. The algorithm of [83] takes $O(n\ log\ n)$ time complexity because it uses sorting, but we improve it to have $O(n)$ time complexity replacing the sorting step with $O(n)$ median search algorithm such as SELECT [66] and present it in Algorithm 2.11. A simplified and more efficient version of $O(n\ log\ n)$ algorithm of [83] is also presented in Algorithm 2.10. Including other two approximate methods [101, 131], we describe all methods in the order of published years.

1990 Pardalos and Kovoor [101] for indefinite case

Their algorithm first departs convex and concave part and underestimates the concave part by piecewise linear convex functions solving 0-1 mixed integer linear program. It guarantees an ε approximate solution. However, the size of mixed integer linear program exponentially grows by 2^N, where N is the number of pieces for the piecewise linear function. Thus, the algorithm gets exponentially deteriorated for the number of variables in N and the level of accuracy.

1991 Moré and Vavasis [83] for strictly concave case

They proposed a method to find a local optimum solution for the strictly concave KSQP and named their algorithm `CKP*`, which stands for Concave Knapsack Problem. We present `CKP*` in Algorithm 2.10 with our modification for efficiency and simplicity. The basic idea is finding three sets $L = \{i : x_i = l_i\}$, $U = \{j : x_j = u_j\}$, and $\{k\} = \{1, 2, ..., n\} \backslash (L \cup U)$ that satisfies

$$-d_i l_i \leq -d_k x_k \leq -d_j u_j$$

as local optimum conditions in (4.21) and (4.22). Note that $-x_k d_k$ is the value of λ because $x_k^* = -\lambda^*/d_k \in [l_k, u_k]$, and $d_i l_i$ and $d_j u_j$ are the slope of separate objective function $f_j(x_j) = \frac{1}{2} d x_j^2$ at l_j and u_j. That is, $-d_j x_j = -\frac{df_j(x_j)}{dx_j} = -f_j'(x_j)$, and it is a kind of efficiency to minimize objective value per a unit of x_j. So the set L is filled with index that has less efficiency than others.

To find the three sets, it first gets the order of index sorting all $-d_j l_j$ in ascending order and sets $x_j = l_j$ for the first half index and $x_j = u_j$ for the last half index except for the median index k as in step 1-3. Then it tests if a feasible solution can be found adjusting x_k within its bound $[l_k, u_k]$. If the current trial solution violates the knapsack constraint, it switches the current half $x_j = u_j$ to l_j or in the reverse way depending on the violated values as step 4.

Algorithm 2.10 $O(n\ log\ n)$; Simplified version of `CKP*` of Moré and Vavasis [83]

```
1. Sort $w = sort\{j : -d_j l_j\ \forall j\}$, where $w(j)$ is the $j-th$ smallest element of $-d_j l_j\ \forall j$
2. Set $k_l$ and $k_u$ ($k_l < k_u$) such as $k_l = 1$ and $k_u = n$
3. Get $k = \left\lfloor \frac{k_l + k_u}{2} \right\rfloor$ and $g = b - \sum_{i=1}^{k-1} l_{w(i)} - \sum_{i=k+1}^{n} u_{w(i)}$
4. Iterate until stopping criterion is satisfied
   If $g < l_k$ (include some $u_j$), set $k_u = k$, $k = \left\lfloor \frac{k_l + k_u}{2} \right\rfloor$, and
        $g = g - \sum_{i=k}^{k_u - 1} l_{w(i)} + \sum_{i=k+1}^{k_u} u_{w(i)}$
   Elseif $g > u_k$ (exclude some $u_j$), set $k_l = k$, $k = \left\lfloor \frac{k_l + k_u}{2} \right\rfloor$, and
        $g = g + \sum_{i=k_l}^{k-1} l_{w(i)} - \sum_{i=k_l+1}^{k+1} u_{w(i)}$
   Else ($l_k \leq g \leq u_k$), finish algorithm with
        $x_j = \begin{cases} l_j & for\ j = w(1), w(2), ...w(k-1) \\ g & for\ j = k \\ u_j & for\ j = w(k+1), w(k+2), ...w(n) \end{cases}$
```

They implemented this binary search type approach with sorting procedure, which dominates the complexity to be $O(n\ log\ n)$. However, the complexity can be down to $O(n)$ simply replacing the sorting procedure to median search procedure such as SELECT [66]. This our improved method is presented in Algorithm 2.11.

Algorithm 2.11 $O(n)$ time improved version of Moré and Vavasis [83]

```
Define a median function: $[L, k, U] = argmed\{s_j : j \in I\}$
where $k$ is the index such that $s_k$ is the $\lceil n/2 \rceil$th $s_j$ for $j \in I$, and $s_i \le s_k \le s_j$ for $i \in L$ and for $j \in U$
  1. Set $\mathbf{s} = \{-d_j l_j \ \forall j\}$ and $U = \{1, 2, ..., n\}$
  2. Find $[L, k, U] = argmed\{s_j : j \in U\}$
  3. Get $g = b - \sum_{j \in L} l_j - \sum_{j \in U} u_j$
  4. Iterate until stopping criterion is satisfied
     If $g < l_k$ ($k^*$ is in $L$), set $x_U = u_U$, $k_{old} = k$, $[L, k, U] = argmed\{s_j : j \in L\}$, and
         $g = g - \sum_{j \in \{k\} \cup U} l_j + \sum_{j \in \{k_{old}\} \cup U} u_j$
     Elseif $g > u_k$ ($k^*$ is in $U$), set $x_L = l_L$, $k_{old} = k$, $[L, k, U] = argmed\{s_j : j \in U\}$, and
         $g = g + \sum_{j \in \{k_{old}\} \cup L} l_{w(i)} - \sum_{j \in \{k\} \cup L} u_j$
     Else ($l_k \le g \le u_k$), finish algorithm with
         $x_j = \begin{cases} l_j & for\ j \in L \\ g & for\ j = k \\ u_j & for\ j \in U \end{cases}$
```

1992 Vavasis [132] for indefinite case

Vavasis extended his study [83] on concave case to indefinite case in [132] as a local solver. Because $x_j(\lambda)$ for $j \in P$ is determined by a λ and at most one x_j for $j \in N$ is the value of $-\lambda/d_j$ at the local optimum as proved in Theorem 4.8, his algorithm essentially finds a solution that satisfies the conditions in (4.21) and (4.22) in Theorem 4.9. So as Algorithm 2.10, it picks the median index $k \in N$ and fixes $x_j = l_j$ or u_j for other index in N. Then, it tests if the knapsack constraint can be satisfied by

$$x_j = \begin{cases} x_j(\lambda) & for\ j \in P \\ -\lambda/d_k & for\ j = k \\ \in [l_j, u_j] & for\ j \in K^Z = \{j \in Z : \lambda = t_j\} \end{cases} \tag{2.24}$$

for $\lambda \in [-d_k l_k, -d_k u_k]$. If no feasible solution is found, it picks a new k as Algorithm 2.10 and tries to find a feasible solution iteratively.

This procedure converges at a local optimum; however, the implementation is not so efficient because he assumes all values of

$$\sum_{j \in P} x_j(\lambda) + \sum_{j \in Z} x_j \tag{2.25}$$

where $x_j = [l_j, u_j]$ for $j \in Z$, are obtained before it starts iteration to find $k \in N$. This means the algorithm should compute all values of (2.25) at all breakpoints of P and Z and uses the values to get a solution in (2.24) for every iteration.

He proposed four versions of algorithms which has complexity of $O(n^3)$, $O(n^2\ log\ n)$, $O(n^2\ log\ n)$, and $O(n(log\ n)^2)$ named by `IKP1`, `IKP1-CG`, `IKP2` and `IKP2-CG` respectively (`IKP` stands for indefinite knapsack problem). `IKP1` and `IKP2` use the straightforward implementation as we described, and `IKP1-CG` and `IKP2-CG` solves the problem with geometric study of Chazelle and Guibas [22] based on `IKP1` and `IKP2`.

Although `IKP1-CG` and `IKP2-CG` are more advanced in complexity, he implemented only `IKP1` and `IKP2` due to the implementation difficulty of [22] and tested the subset-sum problem, which will be discussed in Section 5.2.2. His experiment results show that the time growth is $O(n^2)$ for `IKP1` and $O(n\ log\ n)$ for `IKP2`, but `IKP1` is about 13 times faster than `IKP2` for the size of up to 80. He expected that `IKP2` would outperform over about $n = 2000$. Above all, his two implementations found the global optimum only one time among 80 test problems.

1992 Vavasis [131] for general case

In addition to [132], Vavasis also developed to solve indefinite KSQP approximately in [131]. The study is for the general quadratic programming. It suggests to transform the nonseparable quadratic terms using eigenvalue decomposition to make the problem separable and solves it using his approximation idea. As a special case, he mainly focused on developing algorithms for the indefinite KSQP.

First he showed that strictly concave KSQP can be exactly solved by dynamic programming with exponential time complexity, and then he suggested an approximate dynamic programming idea. The idea is that we can use linear interpolation for sufficiently small intervals on b instead of solving dynamic programming for all b. He calls this *Linearize*, and it leads his algorithm to have polynomial time complexity.

After the algorithm solves approximate dynamic programming for concave part, it solves the convex part with an updated $b = b - \sum_{j \in N} x_j$ using the $O(n\ log\ n)$ sorting method of [50]. So the complexity consists of convex and concave part as

$$O(n_1\ log\ n_1 + n_2 r \alpha(r) log\ r) \tag{2.26}$$

where n_1 is the size of $P \cup Z$, n_2 is the size of N, $\alpha(r)$ is the inverse Ackerman function[10], and r is the number of breakpoints for dynamic programming so r grows exponentially by n_2. Thus, the first part of $(n_1\ log\ n_1)$ in (2.26) is the cost to solve convex part and the second part of $(n_2 r \alpha(r) log\ r)$ is to solve the concave part of indefinite KSQP with the approximate dynamic programming.

The value of [131] is in its conclusion part where he left a possibility that if the problem can be solved dependent on $|log\ \varepsilon|$ - where $\varepsilon \in (0, 1)$ is an approximation parameter - in polynomial time as he suggested, the indefinite quadratic programming may not be *NP-hard* contradicting proofs for *NP-hardness* in [104, 115]. He commented that more proofs are available in [41].

[10]Vavasis [131] noticed that $\alpha(r)$ glows extremely slowly, and it is safe to assume ≤ 6.

2.5. Conclusion

All algorithms that focus on KSQP are considered in this literature review chapter. In the convex case, the various algorithms were discussed with their computational natures and theoretic properties, and we show how the algorithms have been improved through history for over fifty years. In addition, the more robust and efficient version of Newton method is proposed, and the computational superiority of interval test is analytically compared with other methods. This comprehensive survey for convex case leads to develop new hybrid methods in Chapter 4, and the new methods also applied to develop a new tight bound algorithm, `CBS`, for indefinite case. For nonconvex case, the properties that explains the difficulty of the problem is discussed, and all existing algorithms are considered with detailed algorithmic steps. Furthermore, we suggest an idea that improves $O(n\ log\ n)$ time algorithm of Moré and Vavasis [83] to $O(n)$ time algorithm.

3. Indefinite Open-Box Approach to Solution

3.1. Introduction

Since the indefinite case of (P) in (2.2) is an *NP-hard* problem, the solution may not be easily found directly. We use the Lagrangian relaxation of the upper bound constraints of variables x_j, $j \in N$ to generate the following Lagrangian dual problem (D).

$$(D)\ Max_{\boldsymbol{\rho} \geq \mathbf{0}}\ R(\boldsymbol{\rho}) \tag{3.1}$$

where $R(\boldsymbol{\rho})$ with $\boldsymbol{\rho} \in \mathcal{R}^{|N|}$ is

$$\begin{array}{lll} R(\boldsymbol{\rho}) := & Min & \frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{c}^T\mathbf{x} + \sum_{j \in N} \rho_j(x_j - u_j) \\ & s.t. & \sum_{j \in \forall j} x_j = b \\ & & l_j \leq x_j \leq u_j \quad for\ j \in P \cup Z \\ & & l_j \leq x_j \qquad\quad for\ j \in N, \end{array} \tag{3.2}$$

which has open-box constrains ($l_j \leq x_j$ for $j \in N$).

From the Lagrangian duality theorem, see [10], for any $\boldsymbol{\rho} \geq \mathbf{0}$, $R(\boldsymbol{\rho})$ is a guaranteed lower bound on (P). Indeed, the best lower bound is then obtained by solving the Lagrangian dual problem (D). In this chapter, we will use $R(\boldsymbol{\rho})$ as the basis to construct lower and upper bounds of (P) in polynomial time, and the approach is completed by the following three stages:

Stage-1 Develop an efficient algorithm (`OBG`) that finds an exact global optimum solution of $R(\boldsymbol{\rho})$ in $O(n^2)$

Stage-2 Develop a procedure that updates $\boldsymbol{\rho}$ iteratively to solve (D) that generates a lower bound

Stage-3 Develop an upper bounding method, based on the lower bound, to find a feasible solution of high quality in $O(n)$

In section 3.2, we prove that `OBG` solution method has quadratic time worst-case complexity. Then, Stages-2 and 3 are covered in section 3.3. In section 3.4.2, the superior performance of `OBG` is confirmed by experimental comparison with a global solver of Chen and Burer [23], a local solver of Vavasis [132], and the commercial global solver of Cplex 12.6 of IBM using a test-set of problem. The quality and speed of the bounding procedure are also discussed in section 3.4.3.

3.2. Open-Box Global (`OBG`) Solver

To complete the Stage-1, we present a global solver, which is referred to Open-Box Global (`OBG`) solver, in the form of (P'), which has a similar format of (P) in (2.2).

$$\begin{array}{lll}
(P') \quad Min & \frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{c}^T\mathbf{x} & \\
s.t. & \sum_{\forall j} x_j = b & \\
& l_j \le x_j \le u_j & for\ j \in P \cup Z \\
& l_j \le x_j & for\ j \in N
\end{array}$$

It is easily recognized that (P') is bounded if and only if l_j has a finite value for $j \in N \cup Z$; moreover, it also holds for (P).

To the best of author's knowledge, the problem (P') has not been researched as this is the first instance an open-box approach is being utilized to solve the closed-box problem. Lotito (2006, [74]) and Cominetti et al. (2012 [25]) considered the case of $\mathbf{l} \le \mathbf{x}$, but both papers focus only on the strictly convex problems ($d_j > 0\ \forall j$). The indefinite KSQP with the finite valued closed-box constraints ($\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$) has been rarely studied because efficient conditions to verify the global optimality are not known as Sahni [115] and Pardalos and Vavasis [104] proved that the problem is an *NP-hard* problem. Thus, only four previous research, i.e. [83, 101, 131, 132], have presented algorithms, which are considered to be inefficient. [83, 132] find only local optimum solutions, and [101] and [131] find an approximate global optimum in exponential time and polynomial time at a given parameters respectively. In the survey in Section 2.4.1, the above algorithms and their complexity analyses are presented.

Our new method mainly utilizes KKT conditions of (P') based on a two-stage decomposition method, which is inspired by the separability of the problem. A similar idea is employed by Vavasis [132], but the way to treat the two stages is in reverse. Moreover, Vavasis [132] uses KKT conditions with modifications, while our method enumerates all KKT points that satisfies an additional necessary condition of optimization and chooses the best solution. Moreover, our method guarantees to find all global optimum solutions to the problem.

In spite of the exhaustive enumeration procedure, our algorithm is very efficient thanks to the techniques of interval test method having polynomial time complexity and the Lagrangian multiplier shrinking step. The experiment verifies the complexity of $O(|T|\ log\ |T|)$ holds for $n \leq 3000$ with $|T| = 2|P| + |Z|$, and $O\left(|N|(n - |N|)\right)$ holds for greater sizes in the worst case.

3.2.1. Problem characteristics

Observe that (P') can be equivalently expressed as the following two stage model structure:

$$\begin{aligned} (P1) \quad Min \quad & \left(\tfrac{1}{2}\textstyle\sum_{j\in P} d_j x_j^2 - \sum_{j\in Z} c_j x_j\right) + \Phi(\beta) \\ s.t. \quad & \textstyle\sum_{j\in P\cup Z} x_j = \beta \\ & l_j \leq x_j \leq u_j \quad j \in P \cup Z \end{aligned}$$

where

$$(P2) \quad \Phi(\beta) = Min\{\frac{1}{2}\sum_{j\in N} d_j x_j^2 : \sum_{j\in N} x_j = b - \beta,\ l_j \leq x_j,\ j \in N\}$$

We can observe that $(P1)$ has only convex quadratic terms and $(P2)$ has only concave terms in the problems' objective functions sharing a single common variable of β.

The KKT conditions of (P') can be obtained from the KKT conditions of (P) in (2.4) by imposing $\mu_j = 0$ for $j \in N$. These KKT conditions are the necessary conditions for the global optimality of (P') (see page 162 of [10]). `OBG` basically enumerates all candidate solutions based on KKT conditions, but the number of solution to be enumerated can be significantly reduced based on the following necessary condition proved in Theorem 3.1 and Theorem 3.2.

Theorem 3.1. *At a local optimum of $(P2)$, $x_j = l_j$ for $j \in N$ except for at most one $k \in N$ for which $x_k > l_k$.*

Proof. Since the problem $(P2)$ is a strictly concave minimization problem on a convex ployhedron for a given β, its global solution is found at an extreme point of its feasible domain; however because $(P2)$ has a knapsack constraint, it should allow possibly at most one variable to be non extreme point, that is $x_k > l_k$ for a single $k \in N$, as a basic feasible solution. Hence, the theorem is completed. Another approach can be found in Lemma 4 of Vavasis [132] and Theorem 2.1 of Pardalos [99]. Note that the proof of this theorem is presented in the similar Theorem 4.8. □

In addition to Theorem 3.1, Theorem 3.2 proves that two stage problems $(P1)$ and $(P2)$ can have the same Lagrange multiplier for the knapsack constraints.

Theorem 3.2. *Let λ_P and λ_N be optimum Lagrange multipliers of knapsack constraints of $(P1)$ and $(P2)$ respectively as we define λ in (2.4). When $x_k > l_k$ for a single $k \in N$, it holds that*

$$\lambda_P = \lambda_N$$

Proof. Since a single $x_k > l_k$ for $k \in N$ by Theorem 3.1, its value is obtained using the knapsack constraint as

$$x_k = b - \beta - \sum\nolimits_{j \in N \setminus \{k\}} l_j \tag{3.3}$$

and by KKT conditions of $(P2)$, it can also be defined as

$$x_k = -\lambda_N / d_k \tag{3.4}$$

Then, the objective value of $\Phi(\beta)$ is

$$\Phi(\beta) \;=\; \frac{1}{2}\left(\sum\nolimits_{j\in N\setminus\{k\}} d_j l_j^2 + d_k x_k^2\right), \tag{3.5}$$

and by substituting (3.3) in (3.5) and taking partial derivative on β , we have

$$\begin{aligned}\frac{\partial\Phi(\beta)}{\partial\beta} &= -d_k\left(b-\beta-\sum\nolimits_{j\in N\setminus\{k\}} l_j\right)\\ &= -d_k x_k = \lambda_N\end{aligned}$$

due to (3.4). Moreover, if we consider β as a variable, a stationarity KKT condition of $(P1)$ is

$$\frac{\Phi(\beta)}{\partial\beta} - \lambda_P = 0.$$

Hence, it results in

$$\frac{\Phi(\beta)}{\partial\beta} = \lambda_P = \lambda_N,$$

and we conclude the theorem. □

Our solution strategy is enumerating all KKT points that satisfy the necessary conditions proved in Theorem 3.1 and 3.2. Therefore, a global optimum solution is found at $\lambda^* = \lambda_P^* = \lambda_N^*$ when $x_k^* > l_k$ for a single $k \in N$. If $x_j^* = l_j$, $j \in N$, λ_P^* and λ_N^* may not be identical.

Solution characteristics

By the KKT conditions of (P'), we can obtain following solution characteristics for each variable.

$$x_j(\lambda) = \begin{cases} \begin{cases} median\{l_j, u_j, -\lambda/d_j\} \end{cases} & for\ j \in P \\ \begin{cases} l_j & if\ \lambda > c_j \\ u_j & otherwise \end{cases} & for\ j \in Z \\ \begin{cases} l_j & if\ \lambda < -d_j l_j \\ l_j\ or\ -\lambda/d_j & otherwise \end{cases} & for\ j \in N \end{cases} \tag{3.6}$$

The derivation of $x_j(\lambda)$ for $j \in P$ is available in (2.5). For $j \in Z$, $x_j(\lambda)$ is defined based on KKT conditions considering three cases of x_j as

$$\left.\begin{array}{ll} \text{If } x_j = l_j, & \text{then } \mu_j = 0 \text{ and } \lambda \geq c_j \\ \text{If } x_j = u_j, & \text{then } \gamma_j = 0 \text{ and } \lambda \leq c_j \\ \text{If } x_j = (l_j, u_j), & \text{then } \mu_j = \gamma_j = 0 \text{ and } \lambda = c_j \end{array}\right\} \tag{3.7}$$

At an optimum solution, x_j for $j \in Z$ at $\lambda = c_j$ is actually a value within $[l_j, u_j]$ although u_j is assigned for $x_j(\lambda)$ at $\lambda = c_j$. Hence, our algorithm uses $x_j(\lambda)$ only at $\lambda \neq c_j$, and a detailed procedure to get an optimum x_j at $\lambda = c_j$ will be presented in Section 3.2.2.

For $j \in N$, $x_j(\lambda)$ can be derived by considering two cases that x_j is either l_j or strictly greater than l_j as

$$\begin{array}{ll} \text{If } x_j = l_j, & \text{then } \gamma_j \geq 0 \text{ and } \lambda \geq -d_j l_j \\ \text{If } x_j > l_j, & \text{then } \gamma_j = 0 \text{ and } x_j = -\lambda/d_j \end{array}$$

Moreover, by Theorem 3.1, at most one x_j can have the value strictly greater than l_j for an index set of $\{j \in N : \lambda^* > -d_j l_j\}$. $x_j(\lambda)$ for $j \in N$ is illustrated in Figure 3.1, and Figure 4.1a and Figure 4.1b show the behavior of $x_j(\lambda)$ for $j \in P$ and $j \in Z$ respectively.

Note that $x_j(\lambda)\ \forall j$ satisfies all KKT conditions except for knapsack constraint at a given λ because it is derived from KKT conditions without the knapsack constraint.

Definition 3.3. *Breakpoint and λ interval*

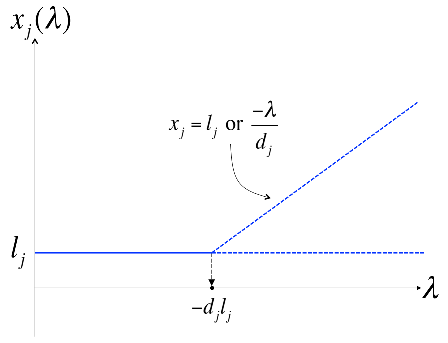


Figure 3.1.: $x_j(\lambda)$ for $j \in N$

Define breakpoints in which $x_j(\lambda)$ is not differentiable. For $j \in P$, $x_j(\lambda)$ have two beak points

$$-d_j u_j \text{ and } -d_j l_j,$$

and as observed in (3.6) each $x_j(\lambda)$ for $j \in Z \cup N$ has one breakpoint

$$t_j = \begin{cases} c_j & for\ j \in Z \\ -d_j l_j & for\ j \in N \end{cases}.$$

Throughout this chapter, we denote T to be the set of finite value breakpoints of P and Z as

$$T = \left\{ \begin{array}{l|l} -d_i u_i, & i \in P,\ u_i < \infty \\ -d_j l_j, & j \in P,\ l_j > -\infty \\ c_k & k \in Z,\ u_k < \infty \end{array} \right\}.$$

The set of breakpoints T is used to construct intervals on λ by sorting elements of T, and we name the intervals λ intervals.

Domain of Lagrange multiplier

As we define $\mathbf{x}(\lambda)$ for solution characteristic, the single Lagrange multiplier λ roles as the key to find optimum solutions, and it can be bounded by KKT conditions and breakpoints.

We denote the maximum finite value of breakpoints by

$$\hat{\lambda}_{max} = \begin{cases} -\infty & if\ P \cup Z = \emptyset \\ max\{T_j\} & otherwise \end{cases}. \tag{3.8}$$

Note that $-d_j u_j$ for $j \in P$ in T are not necessary to get $\hat{\lambda}_{max}$ in practice because it is always $-d_j u_j < -d_j l_j$ by the assumption of $l_j < u_j$. In the case of strictly concave case of (P'), we can simply set $\hat{\lambda}_{max} = -\infty$. Furthermore, since l_j for $j \in N$ is finite value in the bounded problem (P'), the following Lemma 3.4 proves λ^* can be greater than $\hat{\lambda}_{max}$.

Lemma 3.4. *It is possible that* $\hat{\lambda}_{max} \leq t_k < \lambda^*$ *for* $k \in N$

Proof. By contradiction, suppose $\lambda^* < \hat{\lambda}_{max} \leq t_k$ for $k \in N$. For a simple example, consider a (P') such that $l_j \geq 0$ for $j \in P$, $Z = \emptyset$, $|N| = 1$, and $b > \sum_{j \neq k} l_j + l_k$. Then, the global objective value is

$$f(\mathbf{x}^*) = \sum_{j \neq k} d_j l_j^2 + d_k x_k^2$$

since any value of $x_j > l_j$ for $j \in P$ worsens the objective value and it is allowed that $x_k = -\lambda^*/d_k > l_k$ by Theorem 3.1. The value of x_k^* can be obtained by knapsack constraint as

$$x_k = b - \sum_{j \neq k} l_j$$

and since $b > \sum_{j \neq k} l_j + l_k$, the dual solution λ^* is

$$\begin{aligned} x_k = -\lambda^*/d_k &= b - \sum_{j \neq k} l_j > l_k \\ \lambda^* &> -d_k l_k = t_k \end{aligned}$$

Since this is a contradiction, we conclude the lemma. $\square$

Theorem 3.5. *The optimum Lagrange multiplier λ^* is in the range of $[\lambda_{min}, \lambda_{max}]$ where*

$$\begin{aligned} \lambda_{min} &= max\{-\infty,\ t_j : j \in J_u\} \\ \lambda_{max} &= \infty \end{aligned} \tag{3.9}$$

for $J_u = \{j \in Z : u_j = \infty\} \cup N$.

Proof. If $u_j = \infty$ for some $j \in Z \cup N$ by the KKT conditions

$$\begin{aligned} d_j x_j - c_j + \lambda = \gamma_j &\geq 0 \\ \lambda &\geq c_j - d_j x_j \qquad \textit{since } d_j \leq 0 \\ &\geq c_j - d_j l_j = t_j \end{aligned}$$

Thus, $\lambda^* \geq t_j$ for $j \in J_u$, and λ_{min} can be found. Moreover, it is possible that λ^* is greater than any finite breakpoints ($\lambda^* > \hat{\lambda}_{max}$) by Lemma 3.4. Hence, a KKT point can be found at $\lambda^* \in [\lambda_{min}, \infty]$. □

In addition, if the knapsack constraint is inequality as $\mathbf{a}^T\mathbf{x} \leq b$, λ_{min} should be updated to *zero* when the value is negative to ensure $\lambda^* \geq 0$. Moreover, we will show in the next Section 3.2.1.1 that λ_{min} can be set to a greater value if some conditions are satisfied so $[\lambda_{min}, \lambda_{max}]$ can be narrower.

3.2.1.1. Global cases

In this section, we present five global cases that global optimum solutions are found. The cases are determined by the behavior of $x_j(\lambda)$ for $j \in P \cup Z$ when we fix $x_j = l_j$ for $j \in N$. We define it in the knapsack constraint as $g(\lambda)$ with H and G.

$$\begin{aligned} g(\lambda) &= \sum_{j\in N} l_j - b + \sum_{j\notin N} x_j(\lambda) \\ &= \sum_{j\in N} l_j - b + \sum_{j\in L} l_j + \sum_{j\in U} u_j - \lambda \sum_{j\in I}(1/d_j) \\ &= H - \lambda G \end{aligned} \tag{3.10}$$

where

$$L = \{j \notin N : x_j(\lambda) = l_j\},\ U = \{j \notin N : x_j(\lambda) = u_j\},\ I = \{j \in P : l_j < x_j(\lambda) < u_j\}, \tag{3.11}$$

and H is the sum of constant parts and G is the sum of parts that include λ.

Since G is always positive with positive d_j when $I \neq \emptyset$ or *zero* when $I = \emptyset$, $g(\lambda)$ is non-increasing function, and it is a piecewise linear function with discontinuous points at t_j for $j \in Z$ because $x_j(\lambda)$ for $j \in P \cup Z$ has the same properties and $g(\lambda)$ is a sum of $x_j(\lambda)$ for $j \in P \cup Z$ and constants. Then, global cases can be classified with

$$P_l = \{j \in P : l_j = -\infty\} \tag{3.12}$$

$$g_u = g(\lambda_{min}) \tag{3.13}$$

$$g_l = g(\hat{\lambda}_{max}) + \sum\nolimits_{j\in J_Z(\hat{\lambda}_{max})}(l_j - u_j). \tag{3.14}$$

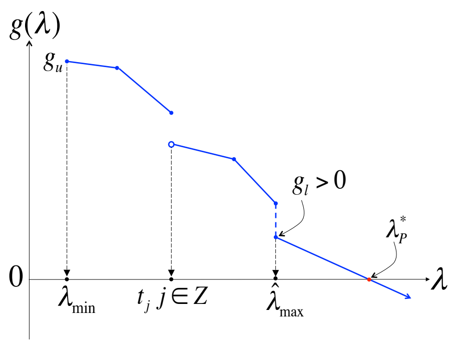
As illustrated in Figure 3.2, g_u and g_l are kinds of two end points of $g(\lambda)$ on $\lambda \in [\lambda_{min}, \hat{\lambda}_{max}]$ with an index set

$$J_Z(\hat{\lambda}_{max}) = \{j \in Z : \hat{\lambda}_{max} = t_j\}$$

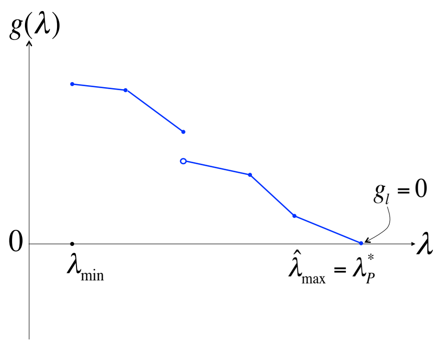
Note that the last summation part of g_l with $J_Z(\hat{\lambda}_{max})$ is to drag down $g(\hat{\lambda}_{max})$ to its minimum value with $x_j \in [l_j, u_j]$ for $j \in Z$.

Global cases

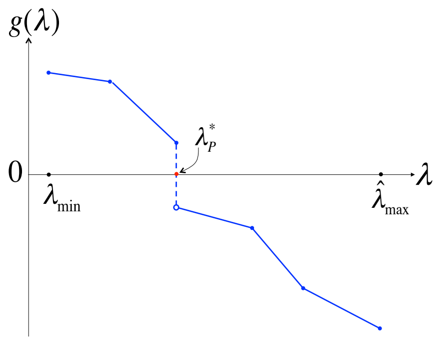
Since $g(\lambda)$ is non-increasing function, we start to determine the global cases of (P') by the sign of g_l and the emptiness of P_l, and then g_u may be used to detect further global cases if it is necessary. We present five global cases in the order from the case that a global optimum solution is determined by smaller operations so the global Case 5 represents the worst case of (P). Examples of each case are illustrated in Figure 3.2.



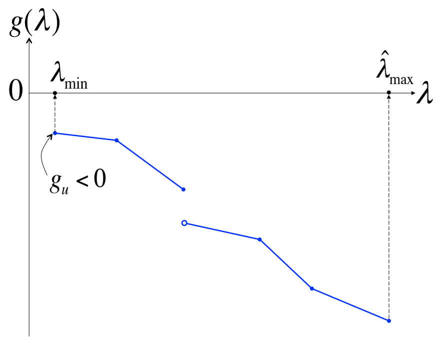
(a) Case 1.2: $g_l > 0$ and $P_l \neq \emptyset$



(b) Case 2: $g_l = 0$



(c) Case 4: $g_l < 0$ and $g_u \geq 0$



(d) Case 5: $g_u < 0$

Figure 3.2.: Global cases

$$\textit{Case } 1. \quad \text{If } g_l > 0 \begin{cases} \textit{Case } 1.1: \ (P) \text{ is infeasible} & \textit{if } P_l = \emptyset \\ \textit{Case } 1.2: \ \lambda^* \geq \lambda_P^* > \hat{\lambda}_{max} & \textit{otherwise} \end{cases}$$

$$\textit{Case } 2. \quad \text{If } g_l = 0 \begin{cases} \textit{Case } 2.1: \ \lambda^* = \lambda_P^* = \hat{\lambda}_{max}, \ \mathbf{x}^* = \mathbf{l} & if \ P_l = \emptyset \\ \textit{Case } 2.2: \ \lambda^* \geq \lambda_P^* = \hat{\lambda}_{max} & otherwise \end{cases}$$

Case 3. If $g_l < 0$ and $\hat{\lambda}_{max} \leq \lambda_{min}$, then $\lambda^* \geq \lambda_{min}$ with a Type B solution

Case 4. If $g_l < 0$ and $g_u \geq 0$, then $\lambda^* \geq \lambda_P^*$

Case 5. If $g_l < 0$ and $g_u < 0$, then $\lambda^* \geq \lambda_{min}$ with a Type B solution

Theorem 3.5 proves that λ_{min} is greater than or equal to the maximum value of breakpoints of N. That means all $x_j(\lambda)$ for $j \in N$ are candidate that can be $x_j > l_j$ at a global optimum; thus, a global optimum solution is either Type A or B by Theorem 3.1.

- Solution Type A: $x_j^* = l_j$ for all $j \in N$
- Solution Type B: $x_j^* = l_j$ for $j \in N\backslash\{k^*\}$ and $x_{k^*}^* > l_{k^*}$ for a single $k^* \in N$

The Type A global solution can be easily found because when x_j for all $j \in N$ are fixed at l_j and (P) becomes a convex problem. The convex problem can be solved via methods[1] in Chapter 2 by searching for a λ where the non-increasing function $g(\lambda)$ passes *zero* or jumps at t_j for $j \in Z$ passing *zero*. We denote the Lagrange multiplier by λ_P^*, and it exists in Case 1.2, 2, and 4. Since $g(\lambda_{max}) < 0$ is guaranteed when $P_l \neq \emptyset$, if $g_l \geq 0$ with nonempty P_l, $g(\lambda)$ must pass *zero* at λ_P^* between $[\hat{\lambda}_{max}, \lambda_{max}]$ as considered in Case 1.2 and 2.2. Case 2.1 simply represent the case when $\mathbf{x}^* = \mathbf{l}$, and Case 4 is detected when the signs of two end points g_l and g_u of $g(\lambda)$ on $\lambda \in [\lambda_{min}, \hat{\lambda}_{max}]$ are different.

In the global solution Type B, $x_{k^*}^*$ is strictly greater than l_{k^*} for a single $k^* \in N$. Then, λ^* is placed where it satisfies

$$\begin{aligned} g(\lambda^*) - l_k + x_k^* &= 0 \quad \textit{since } x_{k^*}^* > l_{k^*} \\ g(\lambda^*) &< 0, \end{aligned}$$

[1]The method Sec+Int in Section 4.3.1.1 is recommended because it is proved to be the best method in speed and reliability by thoughtful experiments in Section 4.4.1.

and if we replace $x_k = l_k$ for a single $k \in N$ with $x_k = -\lambda/d_k > l_k$ in $g(\lambda)$, we have the value of $\hat{g}(\lambda)$

$$\hat{g}(\lambda) = g(\lambda) - l_k + \frac{-\lambda}{d_k}.$$

Hence, $\hat{g}(\lambda)$ is actually a lifted up value from $g(\lambda)$ by $x_k > l_k$, and a KKT point is found when $\hat{g}(\lambda) = 0$ is satisfied. In other words, if $g_l < 0$ or $g(\infty) < 0$ with $P_l \neq \emptyset$, there is a chance that a single $x_k > l_k$ can lift up $\hat{g}(\lambda)$ to touch *zero* as a Type B global solution. Otherwise, if $g_l \geq 0$ and $P_l = \emptyset$ as Case 1.1 and 2.1, the feasibility or a global optimum solution is determined without considering Type B because $g(\lambda)$ cannot be further decreased to give x_k for $k \in N$ a chance to lift up $\hat{g}(\lambda)$.

Our algorithm exploits the technique of interval test method to search for k^* that satisfies $\hat{g}(\lambda) = 0$ for every $k \in N$ in each interval which is constructed by breakpoints of T. This means we have to take $O(|N|)$ time operations in each interval and each point t_j for $j \in Z$, but the enumeration procedure can be saved by constructing the intervals with breakpoints that is greater than

$$\hat{\lambda}_{min} = max\{\lambda_P^*, \lambda_{min}\} \tag{3.15}$$

because KKT points can exist for $\lambda \geq \lambda_{min}$ as proved in Theorem 3.5 and Type B solution can be found only at $\lambda \geq \lambda_P^*$. The idea of narrower λ domain highly decreases the number of intervals to enumerate. As extreme cases, Case 2.1 has one interval as a point $\lambda_P^* = \hat{\lambda}_{max}$ and Case 1.2 and 2.2 have only one interval of $[\lambda_P^*, \lambda_{max}]$.

Another one interval occurs in Case 3. If conditions in Case 1 and 2 are not met, our algorithm obtains λ_{min} to get g_u. But we can detect global Case 3 before g_u is computed. If $\hat{\lambda}_{max} \leq \lambda_{min}$ as a condition of Case 3, there are no breakpoints to construct intervals within $[\lambda_{min}, \lambda_{max}]$, and the bound is the only interval that KKT points are found.

If a global cases is not detected until Case 3, our algorithm executes more computations to get g_u, and it is used to make a decision whether λ_P^* exists. If g_u is negative, we skip Case 4 and consider Case 5 searching for only a Type B global solution because λ_P^* does not exist in Case 5. For the same reason, Case 3 also finds only Type B global solution.

3.2.2. Interval test method and Algorithm

Although our algorithm basically enumerates all KKT points that satisfy Theorem 3.1 to find an exact global optimum, it can be done quite efficiently via techniques in interval test method. This section presents two efficient enumeration procedures at an interval (*interval test*) and at a point (*point test*) of t_j where x_j for $j \in J_Z(t_j)$ can be set to a value in $[l_j, u_j]$. Then, update procedure that requires few operations when we move from an interval to next interval is also addressed in Algorithm 3.2, and the full steps are described in Algorithm 3.1.

Interval test at an interval

Let $[\lambda_l, \lambda_u]$ be the current λ interval that consists of two consecutive breakpoints in T within $[\hat{\lambda}_{min}, \lambda_{max}]$. If $x_j^* = l_j$ for $j \in N$ as Type A global solution and $\lambda_P^* \in [\lambda_l, \lambda_u]$, it can be obtained by

$$\lambda_P^* = H/G$$

with H and G in (3.10), and the objective value of (P) to the corresponding solution is

$$f = \frac{1}{2}\left(R + \sum_{j \in P} d_j x_j^2(\lambda_P^*)\right) - \sum_{j \in Z \setminus J_Z(\lambda_P^*)} c_j x_j(\lambda_P^*) - \lambda_P^* w \tag{3.16}$$

where $R = \sum_{j \in N} d_j l_j^2$, which is $2\Phi(\beta)$ when $x_j = l_j$ for $j \in N$, and

$$w = \begin{cases} 0 & if\ J_Z(\lambda_P^*) = \emptyset \\ b - \sum\limits_{j \in N} l_j - \sum\limits_{j \in P \cup (Z \setminus J_Z(\lambda_P^*))} x_j(\lambda_P^*) & otherwise \end{cases}.$$

Note that essentially $w = \sum_{j \in J_Z(\lambda_P^*)} x_j^*$, and since $\lambda_P^* = c_j$ for $j \in J_Z(\lambda_P^*)$, the last part of (3.16) is expressed with λ_P^* instead of c_j. Thus, Type A global solution can be found at λ_P^*.

To find a Type B global solution with $x_k^* > l_k$ for a single $k \in N$, we can do following efficient enumeration procedure, which is called *interval test*, including a single $k \in N$ into

Algorithm 3.1 $O(|N|(n-|N|))$; Open-Box Global (`OBG`) solver

Given initial values of $f^* = \infty$, $\lambda^* = \infty$, $\lambda_u = \infty$, $f_P = 0$, $w = 0$, $J_Z = \emptyset$, a tolerance ε_{fea}, and sub algorithms 3.3 and 3.2

```
1. Get $H$, $G$, $f_P$ in (3.24) with $P_l$, $\hat{\lambda}_{max}$ (3.8), $g_l$ (3.25), and
   a part of $T = \{-d_i l_i,\ c_j : i \in P,\ j \in Z\}$
2. (Case 1) If $g_l \geq \varepsilon_{fea}$,
   a) (Case 1.1) If $P_l = \emptyset$, finish. $(P)$ is infeasible.
      Else ($P_l \neq \emptyset$)
       i. Get $\lambda_P^* = H/G$ and $f$ in (3.16) $\rightarrow$ Update($f$, $\lambda_P^*$)
      ii. (Case 1.2) Set $\lambda_l = \lambda_P^*$
3. (Case 2) If $|g_l| < \varepsilon_{fea}$,
   a) (Case 2.1) If $P_l = \emptyset$, finish algorithm with $\lambda^* = \hat{\lambda}_{max}$, $\mathbf{x}^* = \mathbf{l}$, and $f^* = f(\mathbf{x}^*)$
      (Case 2.2) Else ($P_l \neq \emptyset$) $\rightarrow$ Update($f$, $\hat{\lambda}_{max}$) and set $\lambda_l = \hat{\lambda}_{max}$
4. (Case 1.2 or 2.2) If $g_l > -\varepsilon_{fea}$, do *interval test* for $f$ in (3.18) $\rightarrow$ Update($f$, $\Lambda_{\hat{k}}$, $\hat{k}$)
   and go to step 12
5. Get $\lambda_{min}$ (3.9)
6. (Case 3) If $\hat{\lambda}_{max} \leq \lambda_{min}$ (and $g_l < 0$),
   a) Set $\lambda_l = \lambda_{min}$ (if $P_l = \emptyset$, get $F_k$ by (3.26))
   b) Do *interval test* for $f$ in (3.18) $\rightarrow$ Update($f$, $\Lambda_{\hat{k}}$, $\hat{k}$) and go to step 12
7. Get $g_u$ in (3.12) and full $T = (\{-d_j u_j : j \in P\}, T)$
8. (Case 4) If $|g_u| < \varepsilon_{fea} \rightarrow$ Update($f$, $\lambda_{min}$, $\emptyset$, $w$, $J_Z(\lambda_{min})$)
   (Case 4) Elseif $g_u > 0$, get $\lambda_P^*$ using Sec+Int $\rightarrow$ Update($f$, $\lambda_P^*$, $\emptyset$, $w$, $J_Z(\lambda_P^*)$) and set
   $\lambda_{min} = \lambda_P^*$
   (Case 5) Else ($g_u < 0$), $(P)$ is in Case 5
9. Select and sort breakpoints in descending order
   $J_T = \{j : \lambda_{min} < T_j\}$ and $[T, J] = sort(T_j : j \in J_T)$
   where $T$ is sorted $T_j$ for $j \in J_T$, and $J$ is corresponding index set of sorted $T$.
10. Include $\lambda_{min}$ into $T$ as $T = (T, \lambda_{min})$ and set $i = 1$, $\lambda_l = T_1$, and $\lambda_u = \infty$
11. Iterate for *point test* and *interval test* with the sub Algorithm 3.2
12. Finish with $f^*$, $\lambda^*$, $k^*$, and $\mathbf{x}^*$ in (3.27)
   a) If $w^* \neq 0$, set $x_j^* = x_j(\lambda^*)$ for $j \in Z \backslash J_Z^*$ and get $x_j^*$ for $j \in J_Z^*$ by Algorithm 3.4
      Else ($w = 0$), set $x_j^* = x_j(\lambda^*)$ for $j \in Z$
```

G and excluding it from H. So the corresponding value of λ for $k \in N$ is computed as

$$\Lambda_k = \frac{H - l_k}{G + 1/d_k} \quad for\ k \in N,$$

and only the selected k in an index set

$$S = \{k \in N : \lambda_l \leq \Lambda_k \leq \lambda_u\}. \tag{3.17}$$

Then, only the index $k \in S$ can lead to find feasible KKT points with $x_k = -\Lambda_k/d_k$. To find the best KKT point at the current interval, we compute $2\Phi(\beta)$ by

$$F_k = r_k + \Lambda_k^2/d_k \quad for\ k \in S$$

with $r_k = R - d_k l_k^2$, and the minimum objective value at the current interval is

$$f = \frac{1}{2}\left(F_{\hat{k}} + \Lambda_{\hat{k}}^2 G + f_P\right) \tag{3.18}$$

where $\hat{k} = argmin\{F_k : k \in S\}$ and

$$f_P = \sum_{j\in L} d_j l_j^2 + \sum_{j\in U} d_j u_j^2 - 2\left(\sum_{j\in L} c_j l_j + \sum_{j\in U} c_j u_j\right) \tag{3.19}$$

is a part of objective values from the fixed solutions in the testing interval, and I, L, and U are index sets used to get H and G in (3.10).

As we can see in Λ_k and F_k, our algorithm actually considers β as a range of

$$g(\lambda_u) \leq \beta + \left(\sum\nolimits_{j\in N} l_j - b\right) \leq g(\lambda_l)$$

rather than a fixed value; therefore, $(P1)$ and $(P2)$ interactively determine a value of β in the range to satisfy

$$\hat{g}(\Lambda_k) = \beta + \left(\sum\nolimits_{j\in N} l_j - b\right) - l_k + x_k = 0$$

lifting up $\hat{g}(\lambda)$ to touch *zero* from negative $g(\lambda)$ with a single $x_k = \frac{-\Lambda_k}{d_k} > l_k$.

***Point test* at a break point t_j for $j \in Z$**

Recall that $x_j(\lambda)$ for $j \in Z$ is set to u_j at $\lambda = t_j$ although it can be a value in $[l_j, u_j]$ as a KKT point. Thus, we have to find KKT points considering $x_j = [l_j, u_j]$ at t_j for $j \in Z$. Since a candidate λ is determined to a t_j in this case, we can find the best KKT point at this point testing which $x_k = -t_k/d_k > l_k$ for $k \in N$ can yield the best objective value, and we name the procedure *point test*.

Suppose we have tested an interval $[\lambda_l, \lambda_u]$, and the λ_l is from t_j in $j \in Z$. Then, we can utilize the current H and G to find KKT points at $\lambda = \lambda_l$ updating H and f_P as

$$\begin{aligned} H &= H - \textstyle\sum_{j \in J_Z(\lambda)} l_j \\ f_P &= f_P + 2 \textstyle\sum_{j \in J_Z(\lambda)} c_j l_j \end{aligned} \tag{3.20}$$

and searching for a selected index set

$$S = \{k \in N : \sum_{j \in J_Z(\lambda)} l_j \le w_k \le \sum_{j \in J_Z(\lambda)} u_j\} \tag{3.21}$$

where

$$w_k = \lambda \left(G + 1/d_k\right) - H$$

is essentially $\sum_{j \in J_Z(\lambda)} x_j$ when a single $x_k = -t_k/d_k > l_k$ is included in the KKT point. Then, a part of objective value is

$$F_k = \frac{1}{2}\left(r_k + \lambda^2/d_k\right) - \lambda w_k$$

where λ is used instead of c_j for $j \in J_Z(\lambda)$, and the minimum objective value at λ is

$$f = \frac{1}{2}\left(f_P + \lambda^2 G\right) + F_{\hat{k}} \tag{3.22}$$

with $\hat{k} = argmin\{F_k : k \in S\}$.

At the first (rightmost) interval

Once intervals are constructed with sorted breakpoints and $\hat{\lambda}_{min}$, *interval test* can be started either at the leftmost interval or rightmost interval as interval test method Algorithm 2.9. We recommend starting from the rightmost interval since it has computational advantages to get initial values and a global optimum can be determined at the rightmost interval in Case 1.2, 2, and 3.

At the first interval we need to calculate initial values of H and G in (3.10) and f_P in (3.19) with three index sets L, U, and I similarly in (3.11). However, if we are at the rightmost interval, because the three index sets are simply

$$L = (P \backslash P_l) \cup Z, \qquad U = \emptyset, \qquad I = P_l \tag{3.23}$$

the initial values can be efficiently calculated without obtaining L, U, and I as

$$H = \sum_{j \notin P_l} l_j - b, \qquad G = \sum_{j \in P_l} 1/d_j, \qquad f_P = \sum_{j \in P \backslash P_l} d_j l_j^2 - 2\sum_{j \in Z} c_j l_j \tag{3.24}$$

and the global case determinant g_l is simply calculated by

$$g_l = H - \hat{\lambda}_{max} G \tag{3.25}$$

Note that since x_j for all $j \in Z$ are set to l_j in (3.23), the last part for g_l in (3.14) is not included in (3.25).

Moreover, if $P_l = \emptyset$, the computational advantage is enlarged at the right most interval since it is simply

$$G = 0 \quad \text{and} \quad H = g_l,$$

and F_k can be easily calculated without Λ_k as

$$F_k = r_k + d_k(l_k - g_l)^2 \quad for\ k \in N \tag{3.26}$$

since $x_k = l_k - g_l$.

Update procedure at the next intervals and a global $\mathbf{x}^*$

If a global optimum is not determined with initial values of (3.24) through Case 1-3 at step 6 of Algorithm 3.1, the initial values are used to test the rightmost interval, and then we moves to the next intervals on the left side to search for a better optimum solution in Case 4 or 5. Interval test technique has strong efficiency in obtaining index set S in (3.17) and (3.21) that are used to enumerate KKT points; moreover, it enhances our algorithm's performance with efficient update procedures for H, G, and f_P as Algorithm 3.2 when we move to the left interval.

Algorithm 3.2 Update procedure in *point* and *interval test*

```
Given $i = 1$, $\lambda_l = T_1$, $\lambda_u = \infty$, and the sub Algorithm 3.3, iterate until termination criteria in step 3 is satisfied.
  1. Do *interval test* for $f$ in (3.18) $\rightarrow$ Update($f$, $\Lambda_{\hat{k}}$, $\hat{k}$)
  2. If $\lambda_l$ is from $t_j$ for $j \in Z$,
      a) Update $H = H - \sum_{j \in J_Z(\lambda_l)} l_j$ and $f_P = f_P + 2\sum_{j \in J_Z(\lambda_l)} c_j l_j$ as (3.20)
      b) Do *point test* for $f$ in (3.22) $\rightarrow$ Update($f$, $\lambda_l$, $\hat{k}$, $w_{\hat{k}}$, $J_Z(\lambda_l)$)
      c) If $i < |T|$, update $H = H + \sum_{j \in J_Z(\lambda_l)} u_j$ and $f_P = f_P - 2\sum_{j \in J_Z(\lambda_l)} c_j u_j$
  3. If $i = |T|$, stop iteration
  4. Update the interval by $\lambda_u = \lambda_l$, $i = i + 1$, and $\lambda_l = T_i$
  5. If $\lambda_l$ is from $-d_j l_j$ for $j \in P$,
      a) Update $H = H - l_j$, $G = G + 1/d_j$, and $f_P = f_P - d_j l_j^2$
     Else ($\lambda_l$ is from $-d_j u_j$ for $j \in P$),
      b) Update $H = H + u_j$, $G = G - 1/d_j$, and $f_P = f_P + d_j u_j^2$
```

Algorithm 3.2 first does *interval test* for the given rightmost interval at step 1, and then check if λ_l, the lower bound of the current interval, is t_j for $j \in Z$ to decide whether the algorithm needs to do *point test* at step 2.b. Since x_j for $j \in J_Z(\lambda_l)$ can be set to any value in its range $[l_j, u_j]$ from the fixed l_j when we move on λ_l, H and f_P are updated as (3.20) at step 2.a for *point test*, and then H and f_P are updated at step 2.c after *point test* for *interval* *test* at the left next interval since x_j for $j \in J_Z(\lambda_l)$ are fixed to u_j for all left intervals. The interval $[\lambda_l, \lambda_u]$ are updated at step 4 for the next interval, and the algorithm tries to find a better optimum solutions at the interval going back to step 1.

If λ_l is from $-d_j l_j$ for $j \in P$, x_j is switched from l_j to a value of $-\lambda/d_j$ in the next interval. So H, G, and f_P are updated to reflect the status change of x_j at step 5.a. Similarly if λ_l is from $-d_j u_j$ for $j \in P$, since x_j is fixed to u_j from $-\lambda/d_j$, the values of H, G, and f_P are updated at step 5.b.

Algorithm 3.3 $O(1)$; `Update`: a sub function to update the best values

```
Update($f$, $\lambda$, $k$, $w$, $J_Z$) $\rightarrow$ $f^*$, $\lambda^*$, $k^*$, $w^*$, $J_Z^*$
If $f < f^*$,
    1. Update $f^* = f$ and $\lambda^* = \lambda$
    2. If $k = \emptyset$, update $k^* = \emptyset$; Else update $k^* = k$
    3. If $w = \emptyset$, update $w^* = 0$; Else update $w^* = w$, and $J_Z^* = J_Z$
```

Whenever the best objective value f is selected from an index set S at an interval or a point, it is compared with the best objective value f^*, and if the new value f is better than best value f^*, a sub function `Update` in Algorithm 3.3 is used to update not only objective value but also all necessary values to obtain a global optimum solution at the final step 12 in the main Algorithm 3.1 as

$$x_j^* = \begin{cases} x_j(\lambda^*) & for\ j \in P \cup (Z \backslash J_Z^*) \\ l_j & for\ j \in N \backslash k^* \\ -\lambda^*/d_{k^*} & for\ j = k^* \end{cases} \tag{3.27}$$

For $j \in J_Z^*$, if $w = 0$, we conclude $x_j^* = l_j$ for all $j \in Z$; otherwise the value of w can be distributed to x_j^* keeping it within $[l_j, u_j]$ in any preferred way, and Algorithm 3.4 shows an easy linear time way.

Since it is well known that KKT conditions are necessary conditions for indefinite problem

Algorithm 3.4 $O(|J_Z^*|)$; Get x_j^* for $j \in J_Z^*$ distributing w

```
Given $J_Z^* \subseteq Z$, $w^*$, and $i = 1$
  1. Iterate to distribute $w^*$ to $x_j^* \in [l_j, u_j]$ for $j \in J_Z^*$
      a) Set $j = J_Z^*(i)$, $x_j^* = max(l_j, u_j - w^*)$, and $w^* = w^* - u_j + x_j^*$
      b) If $w^* = 0$, finish iteration; Else $i = i + 1$
```

(P), one of KKT points is the global optimum solution. All KKT points that satisfy theorem 3.1 are enumerated very efficiently through *interval test* and *point test* in Algorithm 3.1 utilizing the idea of interval test method. Therefore, Algorithm 3.1 guarantees a global optimum solution, and if it is necessary, we can find all global optimum solutions keeping the solutions that result in the same best objective value. Moreover, in the next section, we show that the algorithm has at most quadratic complexity in the worst case even though it is an enumerative algorithm.

3.2.2.1. Computational complexity; $O\left(n^2\right)$

For the worst case complexity, we may ignore procedures to update H, G, and f_P in Algorithm 3.2 and to update best values in Algorithm 3.3 because the procedures have only $O(1)$ or $O(|Z|)$ complexity. The cost to get λ_P^* for global Case 4 is $O(n)$ time if the median search method in Algorithm 2.3 is used. Other minor operations are consumed to get initial values of H, G, and f_P, required values such as T, λ_{min}, $\hat{\lambda}_{max}$, and g_l, and a global optimum solution in (3.27) and Algorithm 3.4; all the operations are taken only one time with the complexity of $O(n)$.

A main cost occurs when our algorithm enumerate KKT points in *point test* and *interval test* since the both procedures takes $O(|N|)$ complexity at each t_j for $j \in Z$ and λ intervals. In the worst case, the maximum number of enumeration steps is $2(|P| + |Z|)$ because of $|T| = 2|P| + |Z|$ number of *interval tests* and $|Z|$ number of *point tests*. Therefore, the enumeration steps requires the complexity of

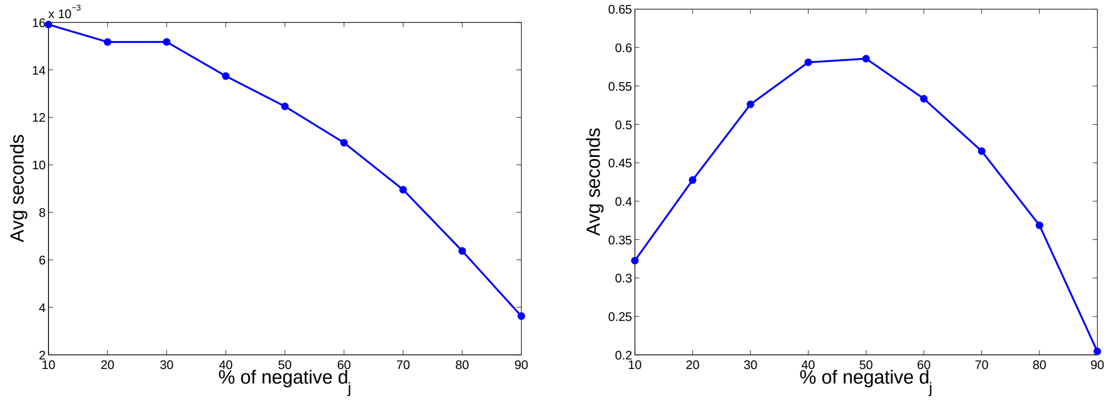
$$O\left(|N|(|P| + |Z|)\right) = O\left(|N|(n - |N|)\right) \tag{3.28}$$

Another main cost occurs in step 9 of the main Algorithm 3.1 to sort breakpoints T. So if all breakpoints are greater than λ_{min} in the worst case, the sorting step spend the complexity of

$$O(|T|\ log\ |T|) \tag{3.29}$$

Thus, the worst case occurs in global Case 5 because it has the largest number of breakpoints to sort and the largest number of intervals and points to enumerate.

In conclusion, the worst case complexity of our algorithm is $O(n^2)$ by (3.28) as $n \to \infty$ and $|N| \to n/2$. However, sorting cost dominates the enumeration steps in practice for the small size problems. In our experiment environment that will be described in Section 3.4, the complexity is dominated by the sorting procedure for size of $n \leq 3000$ as shown in Figure 3.3.



(a) $O(|T|\ log\ |T|)$ at $n = 10^3$ (b) $O\left(|N|(n - |N|)\right)$ at $n = 10^4$

Figure 3.3.: Complexity curves on the proportion of $|N|$

The Figures are drawn with the average time of 10 small coefficient problems ($I1$) and 10 large coefficient problems ($I2$), which are described in the next paragraph, in global Case 5 (worst case) problems with $|Z| = 0\%$ and various proportions of $|N|$ from 10% to 90% at each size of $n = 10^3$ and $n = 10^4$. If the complexity is dominated by $O\left(|N|(n - |N|)\right)$ (3.28), the average time should have a peak at 50% as Figure 3.3b in the problems of $n = 10^4$. However, the peak is at 10% in the problem of $n = 10^3$ as Figure 3.3a. It is because the sorting step spends more CPU time to sort breakpoints of positive d_j than enumeration step. Moreover, the practical CPU time is much less in global Case 1, 2, and 3 because the cases have only one interval; thus, the complexity is dominated by the minor operations and has $O(n)$ time complexity.

Two random problems: $(I1)$ **and** $(I2)$

Two indefinite random problems are generated based on the settings of

$(I1)$ Small coefficient problem: $d_j = [1, 10]$, $c_j = 0$, $a_j = 1$, $l_j = [-10, 0]$, $u_j = [0, 10]$

$(I2)$ Large coefficient problem: $d_j = [1, 2000]$, $c_j = 0$, $a_j = 1$, $l_j = [-1000, 0]$, $u_j = [0, 1000]$

All coefficients are generated by MATLAB's built-in uniform random number generator `rand.m`, and an index set N for $d_j < 0$ is randomly generated by MATLAB's built-in function `randperm.m` in the size of the given proportion of negative d_j. Then, the indefinite problems are generated by changing the sign of d_j for $j \in N$ to negative, and u_j for $j \in N$ is set to the ∞ by `inf` in MATLAB. We set both indefinite problems have $c_j = 0$ and $a_j = 1$ because any (P) can be transformed to have the coefficients by transformation in Appendix A.2, and, above all, the format gives good base problems to generate Case 5 problems. Once $(I1)$ and $(I2)$ are generated, we change $l_j \leftarrow 2l_j$ for $j \in N$ to generate Case 5 problems that satisfies $\lambda_{min} < min\{T_j\}$.

Right hand side b is also controlled to generate the intended problems by

$$b = b_{min} + \beta(\hat{b}_{max} - b_{min})$$

where $b_{min} = \sum_{\forall j} l_j$ and $\hat{b}_{max} = \sum_{j \in N} l_j + \sum_{j \in P} u_j$. Since problems are feasible only when $g_l \leq 0$, b should satisfy

$$g_l = b_{min} - b \leq 0 \rightarrow b_{min} \leq b,$$

and Case 4 can happen if we set b to be

$$g_u = \hat{b}_{max} - b \geq 0 \rightarrow b \leq \hat{b}_{max}$$

Thus, we used $\beta = 1.2$ for Case 5 to ensure that $g_u < 0$.

3.3. Bounding with Open-Box Global (OBG) solver

This section explains the procedure to find a lower bound by iteratively solving $R(\boldsymbol{\rho})$ with OBG at updated $\boldsymbol{\rho}$ for the dual problem (D) in (3.1). Then, a heuristic method is developed to find a feasible upper bound solution based on the lower bounding solution.

3.3.1. Lower bounding procedure

For any $\boldsymbol{\rho} \in \mathcal{R}^{|N|}$, $R(\boldsymbol{\rho})$ is an open-box problem of the type (P), and it is efficiently solvable to global optimality by the OBG solver in Algorithm 3.1. Furthermore, since $R(\boldsymbol{\rho})$ is a piecewise concave function in a vector $\boldsymbol{\rho}$ due to duality, there exists $\boldsymbol{\rho}^* \geq \mathbf{0}$ that maximizes $R(\boldsymbol{\rho})$ in (D). So $R(\boldsymbol{\rho}^*)$ guarantees the lower bound of (P). However, finding $\boldsymbol{\rho}^*$ is time-consuming since a search algorithm, such as a subgradient optimization procedure, must be applied. To compute quick lower bounds, we consider the restricted dual function $\hat{R}(w)$ that uses some univariate $w \geq 0$ for ρ_j, i.e. $\rho_j = w$ for $j \in N$. Then, it also gives a lower bound, and it is trivial that

$$R(\boldsymbol{\rho}^*) \geq \hat{R}(w).$$

Since $\hat{R}(w)$ is a piecewise concave function, the Lagrangian multiplier w^* can be obtained by a root-finding method such as bisection search, which has a finite iteration of $\left\lceil log_{0.5}\left(\frac{\varepsilon_{gap}}{\lambda_{max}-\lambda_{min}}\right)\right\rceil$ with the tolerance ε_{gap}.

The subgradient of $\hat{R}(w)$ with respect to w is $\nabla_w \hat{R}(w) = \sum_{j\in N}(x_j - u_j)$. So w^* is found with a bisection point in the range of $[w_L, w_U]$ such that $\nabla_w \hat{R}(w_L) \geq 0 \geq \nabla_w \hat{R}(w_U)$. Note that if $\nabla_w \hat{R}(0) < 0$, the maximum value of $\hat{R}(w)$ is simply at $w^* = 0$ since $\nabla_w \hat{R}(w) < 0$ for $w \geq 0$.

Let the solution of $\hat{R}(w)$ be $\mathbf{x}^w$. If $x_j^{w*} \leq u_j$ for $j \in N$, then, $\mathbf{x}^{w*}$ is a global optimum solution of the primal problem (P). Otherwise, there exists $k \in N$ such that $x_k^{w*} > u_k$ and

$x_j^{w*} = l_j$ for $j \in N \backslash \{k\}$. Then, the lower bound is given by

$$LB^* := \frac{1}{2}(\mathbf{x}^{w*})^T \mathbf{D}\mathbf{x}^{w*} - \mathbf{c}^T\mathbf{x}^{w*} + w^* \sum_{j \in N}(x_j^{w*} - u_j). \tag{3.30}$$

3.3.2. Upper bounding procedure

Note that if $x_j^w \leq u_j$, $j \in N$, then, $\mathbf{x}^w$ is a global optimum solution of (P). Otherwise, we construct a feasible upper bound solution $\hat{\mathbf{x}}$ by utilizing x^w and the Lagrange multiplier λ^w of the knapsack constraint in $\hat{R}(w)$. The basic idea is to try to find $\hat{\mathbf{x}}$ that satisfies the part of KKT conditions of (P). Consider a part of KKT conditions of (P) for $j \in N$:

$$\begin{gathered} d_j x_j + \lambda + \mu_j - \gamma_j = 0 \\ \mu_j(x_j - u_j) = 0, \;\; \gamma_j(l_j - x_j) = 0 \\ \mu_j \geq 0, \;\; \gamma_j \geq 0 \end{gathered} \tag{3.31}$$

where λ, μ_j, and γ_j are Lagrangian multipliers for constraints of $\sum_{\forall j} x_j = b$, $x_j \leq u_j$, and $l_j \leq x_j$. Then, $\hat{\mathbf{x}}$ that satisfies 3.31 can be obtained with λ^w and $k \in \{j \in N : x_j^w > l_j\}$. In the sense of Theorem 3.1, we allow only one x_k to be a value within $[l_k, u_k]$. Then, since KKT conditions (3.31) implies

$$x_j = \begin{cases} l_j & if \;\; d_j u_j + \lambda > 0 \\ u_j & if \;\; d_j l_j + \lambda < 0 \\ l_j, u_j, \;\; or \;\; -\lambda/d_j & otherwise \end{cases},$$

the upper bound solution for k is set to

$$\hat{x}_k = \begin{cases} l_k & if \;\; d_k u_k + \lambda^w > 0 \\ u_k & if \;\; d_k l_k + \lambda^w < 0 \\ -\lambda^w/d_k & otherwise \end{cases}. \tag{3.32}$$

For other variables for $j \in N \backslash \{k\}$, KKT conditions (3.31) also implies

$$\text{If } x_j = l_j, \quad \text{then } \mu_j = 0 \text{ and } \gamma_j = d_j l_j + \lambda \geq 0$$
$$\text{If } x_j = u_j, \quad \text{then } \gamma_j = 0 \text{ and } -\mu_j = d_j u_j + \lambda \leq d_j l_j + \lambda \leq 0.$$

Thus, a feasible solution is constructed for $j \in N \backslash \{k\}$ with

$$\hat{x}_j = \begin{cases} l_j & if\ d_j l_j + \lambda^w \geq 0,\ j \neq k \\ u_j & otherwise,\ j \neq k. \end{cases}$$

Then, the knapsack constraint is considered with x_j^w for $j \notin N$ and $\hat{x}_j$ for $j \in N$. If the solution satisfies the knapsack constraint, i.e.

$$\sum_{j \in P \cup Z} x_j^w + \sum_{j \in N} \hat{x}_j = b,$$

it is a feasible upper bound solution. However, if it is not satisfied, we find a solution x_j for $j \in P \cup Z$ to be feasible to the knapsack constraint by solving the following convex KSQP

$$\begin{aligned} f_c := \quad Min \quad & \tfrac{1}{2} \textstyle\sum_{j \in P} d_j x_j^2 - \sum_{j \in Z} c_j x_j \\ s.t. \quad & \textstyle\sum_{j \in P \cup Z} x_j = \hat{b} \\ & l_j \leq x_j \leq u_j \quad for\ j \in P \cup Z \end{aligned}$$

with a given $\hat{b} = b - \sum_{j \in N} \hat{x}_j$ using the algorithms in Chapter 2. In the case that f_c is not feasible because $\hat{b}$ does not satisfy

$$\sum_{j \in P \cup Z} l_j \leq \hat{b} \leq \sum_{j \in P \cup Z} u_j,$$

some $\hat{x_j}$ for some $j \in N$ are adjusted from l_j to u_j or vice versa. A proposed adjustment procedure is described in step 6 and step 7 in Algorithm 3.5.

Algorithm 3.5 Upper Bounding algorithm with OBG

```
Let the objective function value of $(P)$ be $f(x) = \frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{c}^T\mathbf{x}$
 1. Given $w$, solve $\hat{R}(w)$ by OBG to get $x^w$, $\lambda^w$, $LB = f(\mathbf{x}^w)$, and $k = \{j \in N : x_j^w > u_j\}$
 2. If $\{k\} = \emptyset$, Stop with $UB = f(x^w)$
 3. Get $t_j = d_j l_j + \lambda^w$ for $j \in N \backslash \{k\}$, $L = \{j : t_j \geq 0\}$, and $U = \{j : t_j < 0\}$
 4. Set $x_j = l_j$ for $j \in L$, $x_j = u_j$ for $j \in U$, and $x_k$ by (3.32)
 5. Get $b_{\min} = \sum_{j \notin N} l_j$, $b_{\max} = \sum_{j \notin N} u_j$, and $\hat{b} = b - \sum_{j \in N} x_j^w$
 6. Iterate while $\hat{b} < b_{\min}$
      $r = argmax\{t_j : j \in U\}$, $U = U \backslash \{r\}$, $x_r = l_r$, $\hat{b} \leftarrow \hat{b} + u_r - l_r$
      If $\hat{b} > b_{\max}$, set $\hat{b} \leftarrow \hat{b} - u_r + l_r$, $l_r \leftarrow \hat{b} + u_r - b_{\max}$, $u_r \leftarrow \hat{b} + u_r - b_{\min}$, and go to step 9
 7. Iterate while $\hat{b} > b_{\max}$
      $r = argmin\{t_j : j \in L\}$, $L = L \backslash \{r\}$, $x_r = u_r$, $\hat{b} \leftarrow \hat{b} - u_r + l_r$
      If $\hat{b} < b_{\min}$, set $\hat{b} \leftarrow \hat{b} + u_r - l_r$, $l_r \leftarrow \hat{b} + l_r - b_{\max}$, $u_r \leftarrow \hat{b} + l_r - b_{\min}$, and go to step 9
 8. Solve $f_c$ with $\hat{b}$ to get $x_j$ for $j \notin N$; Stop with $x_j = \hat{x}_j$ for $j \in N$ and $UB = f(x)$
 9. Solve the indefinite problem with $\hat{b}$ to get $x_j$ for $j \in P \cup Z \cup \{r\}$
      a) Set $\hat{x}_r = u_r$, $x_j = l_j$ *for* $j \in P \cup Z$. Set $x_j = \hat{x}_j$ for $j \in N$ and $UB_1 = f(x)$
      b) Solve OBG in $\lambda \in [-d_r l_r, -d_r u_r]$ to get $x_j$ for $j \in P \cup Z \cup \{r\}$. Set $x_j = \hat{x}_j$ for $j \in N \backslash \{r\}$ and $UB_2 = f(x)$
      c) Finish with $UB = Min\{UB_1,\ UB_2\}$
```

Step 9 in Algorithm 3.5 considers the case when the adjustment procedure for $\hat{b}$ fails in step 6 and step 7. In other words, step 6 fails when there does not exist r such that

$$\hat{b} + u_r - b_{\max} \leq x_j \leq \hat{b} + u_r - b_{\min},$$

and step 7 fails when there does not exist r such that

$$\hat{b} + l_r - b_{\max} \leq x_r \leq \hat{b} + l_r - b_{\min}.$$

For the either one of two cases, step 9 finds a feasible solution considering the cases that $x_r = u_r$ in step 9.a and $x_r \in [l_r, u_r)$ in step 9.b.

The restricted dual function $\hat{R}(w)$ is guaranteed to converge to the lower bound LB^* in (3.30) at the final iteration of the bisection search method; however, the upper bound solution obtained by Algorithm 3.5 with w^* has no guarantee of the best upper bound UB^*. Thus, we compute the upper bound UB at every iteration during $\hat{R}(w)$ is solved, and the minimum value of upper bound is picked for UB^*.

A random problem GN2($n = 100,\ \delta = 10,\ |N| = 20,\ |Z| = 16$), which will be described in Section 3.4.3, is used to illustrate the progression of LB and UB in Figure 3.4. As iteration goes, LB converges to LB^*, while UB fluctuates in a certain range. As shown in Figure 3.4, for GN2 with various parameters, LB^* is weak; however, UB^* is generally quite close to the global optimum. UB^* in the illustrated example the has relative gap $(= (UB^* - obj(Global))/|obj(Global)|)$ of 1.17%.

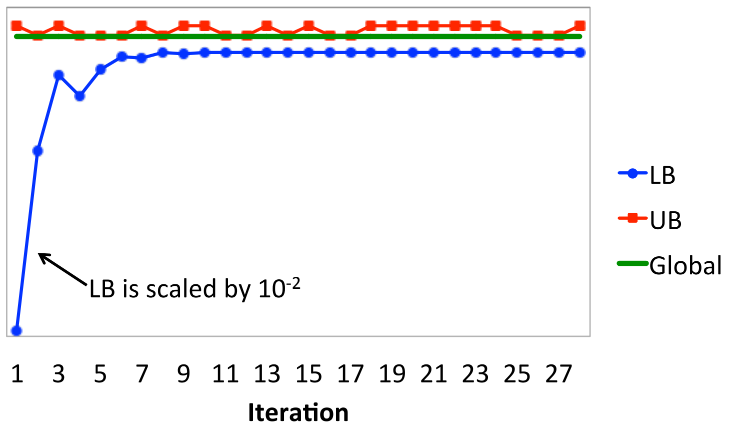


Figure 3.4.: Behavior of lower and upper bounds by iteration

3.4. Experiments

This section is to test the performance of OBG and OBG-based bounding procedure. A local and two global solvers are selected to compare with our proposed methods in the respect of speed and solution quality. All experiments and implementations are done in MATLAB 2013a (version 8.1) with two identical computers (Intel core i5, 3.47Ghz, 8GB RAM, WINDOWS 7, 64bit). The codes are generated by all MATLAB's features such as Just-in-Time compilation, copy-on-write type editing, and fast full vector computation (see Appendix B.2 for an efficient coding style in MATLAB). For the time measurement, the stop watch like function `tic`/`toc` is used rather than `cputime` that counts the time that CPU are actually used by WINDOWS operations (see Appendix 4.3.2 for more detail).

Implementation

In the implemented MATLAB code, the secant method of Dai and Fletcher (2006, [31]) is used to find λ_P^* in the global Case 4 because their experiments and the experiments in Section 4.4.1 show that the secant method is superior to $O(n)$ time median search method. The secant method narrows the domain of λ iteratively by secant steps until either the violation value $|g(\lambda)| > \varepsilon_{fea}$ or the domain is greater than 0.001. Then, if there exist c_j, $j \in Z$ within the narrowed domain, it sorts the selected c_j and does the interval test like steps to find λ_P^*. Otherwise, it keeps iteration with secant steps. Note that the feasibility tolerance is set to

$$\varepsilon_{fea} \quad = \quad max\{1e-7, \varepsilon(n+|b|)\}$$

in the same reason described in Section 4.4 with the machine precision $\varepsilon = 2.2204e-16$.

The Algorithm 3.1 requires to sort breakpoints in step 9. We selected MATLAB's built-in function `sort.m` because it implements QUICKSORT algorithm which is known as the fastest sorting algorithm in practice as described in Appendix B.1.

Compared solvers

In order to compare the performance of `OBG` and the bounding procedure, three different solvers are used.

- `VAV92`: `IKP1` of Vavasis (1992, [132])

  A local solver `IKP1` ($O(n^3)$ complexity) of Vavasis [132] is used, and further reasons to choose it over other versions of solvers in Vavasis [132] is available in Appendix A.5.2. Because VAV92 works only when finite values of u_j for $j \in N$ exist, we give $u_j = \frac{1}{a_j}\left(b - \sum_{i \in N, i \neq j} a_i l_i\right)$, which does not affect to the optimal solution.

- `CHB12`: Chen and Burer (2012, [23])

  A recent version of global quadratic programming solver of Chen and Burer (2012, [23]) is compared. The code is obtained in Burer's website[2], and options described in Appendix A.5.1 are used to avoid time consuming warning messages.

- `CPX-G`: `cplexqp.m` of CPLEX 12.6

  The recently (December 2013) released version of global quadratic programming solver of IBM CPLEX 12.6 is used to compare the solution quality and speed with our proposed methods.

More detailed explanation and options that are used in the experiments are available in Appendix A.5.

3.4.1. Random test problems

A set of randomly generated problem is used to test the performance of the proposed algorithms with four parameters:

- n: the total number of variables
- δ: the size of problem data
- ζ: the proportion of $d_j = 0$ out of n. i.e. $|Z| = \lfloor \zeta n \rfloor$
- ν: the proportion of $d_j < 0$ out of $n - |Z|$. i.e. $|N| = \lfloor \nu(n - |Z|) \rfloor$

As we did for test problems $(I1)$ and $(I2)$ in Section 3.2.2.1, MATLAB's built-in uniform random number generator **`rand.m`** is used for problem data, and **`randperm.m`** is used to generate Z and N. We name the test problem $GN1(n, \delta, \zeta, \nu)$. The steps to generate $GN1$ is followed.

[2]http://dollar.biz.uiowa.edu/~sburer/pmwiki/pmwiki.php%3Fn=Main.QuadprogBB%3Faction=logout.html

Random problem generator: `GN1`(n, δ, ζ, ν)

1. Generate random data

   $d_j \in [1, \delta]$, $c_j \in [0, \delta^2]$, $a_j \in d_j + [0, \delta]$, $l_j \in -1 - [0, \delta^2]$, $u_j \in 1 + [0, \delta]$

2. Generate a random solution $x_j \in [l_j, u_j]$ to construct $b = \sum_{\forall j} a_j x_j$

3. Generate Z in the size of $\lfloor \zeta n \rfloor$, and set $d_j = 0$ for $j \in Z$

4. Generate N in the size of $\lfloor \nu(n - |Z|) \rfloor$ from $\{1, ..., n\} \backslash Z$, and $d_j = -d_j$ for $j \in N$

5. Set $u_j = +\infty$ for $j \in N$

Note that the lower limit l_j is set to a considerably small negative value. This generates a small enough λ_{min} to construct considerably many intervals in λ by breakpoints of $d_j \geq 0$. Thus, $GN1$ is designed to be disadvantageous to `OBG` solver.

3.4.2. Experiments using `OBG`

The initial computational experiments of `OBG` is conducted in the various choices of (n, δ, ζ, ν): $\zeta = 0$, $\nu = 0.5$, $\delta = 10$ and 100, and 3 size groups of small (n from 50 to 500), medium (n from 1,000 to 20,000), and large (n from 30,000 to 150,000). The value of $\zeta = 0$ means that `OBG` does not do the point test, and the combination of $\zeta = 0$ and $\nu = 0.5$ results in the worst case problem for `OBG` as mentioned in Section 3.2.2.1. 10 random problems are tested for each instance, and the time limit of 30 minutes are give for all solvers.

The test results are summarized with average CPU time and coefficient of variation (cov in parenthesis). The solution quality of local solver `VAV92` is measured by the relative error metric:

$$\text{rel. error} = [\text{obj}(\texttt{VAV92})\text{-obj}(\texttt{OBG})]/|\text{obj}(\texttt{OBG})| \times 100\%$$

The results of the experiments are in Table 3.1, 3.2, and 3.3 for each size group. Through all sizes, `OBG` finds a global solution significantly faster than all other solvers. In small size group (Table 3.1), `CHB12` is considerably slow or unable to solve within 30 minute time limit. So CHB12 is excluded for larger size group experiments. The time growth rate of local solver VAV92 is very high, and it consumes slightly less time (about 29 minutes) than the 30 minute time limit. Thus, for large size problems, CPX-G is only testable with OBG, but it is also significantly slower than OBG.

Table 3.1.: Average CPU times for small problems using GN1$(n, \delta, 0, 0.5)$

Solver name	Data size parameter	Solution details[1]	Problem size, n 50	80	100	250	500
OBG	$\delta = 10$	cpu sec	0.00064	0.00064	0.00051	0.00088	0.00154
		(cov)[2]	(0.303)	(0.503)	(0.099)	(0.142)	(0.213)
	$\delta = 100$	cpu sec	0.00060	0.00062	0.00064	0.00110	0.00203
		(cov)	(0.391)	(0.209)	(0.111)	(0.157)	(0.155)
CPX-G	$\delta = 10$	cpu sec	0.012	0.012	0.012	0.018	0.057
		(cov)	(0.117)	(0.111)	(0.067)	(0.043)	(0.789)
	$\delta = 100$	cpu sec	0.012	0.015	0.027	0.040	0.112
		(cov)	(0.295)	(0.375)	(0.708)	(0.507)	(0.468)
CHB12	$\delta = 10$	cpu sec	250.37	714.54	*[3]	*	*
		(cov)	(0.996)	(0.520)			
	$\delta = 100$	cpu sec	*	*	*	*	*
		(cov)					
VAV92	$\delta = 10$	cpu sec	0.007	0.018	0.025	0.112	0.430
		(cov)	(0.050)	(0.505)	(0.506)	(0.133)	(0.043)
		rel. error	72.5%	74.6%	41.4%	57.6%	61.3%
	$\delta = 100$	cpu sec	0.008	0.014	0.021	0.108	0.4237
		(cov)	(0.189)	(0.017)	(0.007)	(0.017)	(0.007)
		rel. error	88.2%	82.2%	91.5%	90.6%	70.0%

1. CPU seconds and relative errors are averaged over 10 random problems
2. Coefficient of variation, cov=std.deviation/avg cpu time
3. 30 minutes cpu time limit or memory exceeded for all problems

Sensitivity of OBG solver

The sensitivity of OBG to the various proportions of ζ and ν at $\delta = 10$ is measured in n from 10,000 to 100,000. The average cpu times of 10 runs of each instance are in Table 3.4 and 3.5. The relative efficiency of OBG relative CPX-G is measured by the 'speed-up factor', defined by the logarithmic measure,

$$\mathtt{SF} = ln \frac{cpu - time(\mathtt{CPX-G})}{cpu - time(\mathtt{OBG})}$$

Table 3.2.: Average CPU times for medium problems using GN1$(n, \delta, 0, 0.5)$

Solver name	Data size parameter	Solution details	Problem size, n 1,000	5,000	10,000	15,000	20,000
OBG	$\delta = 10$	cpu sec	0.003	0.030	0.092	0.167	0.205
		(cov)	(0.148)	(0.195)	(0.355)	(0.300)	(0.338)
	$\delta = 100$	cpu sec	0.005	0.058	0.208	0.449	0.639
		(cov)	(0.082)	(0.123)	(0.147)	(0.206)	(0.207)
CPX-G	$\delta = 10$	cpu sec	0.143	2.342	7.297	16.535	27.809
		(cov)	(1.025)	(0.152)	(0.209)	(0.190)	(0.200)
	$\delta = 100$	cpu sec	0.338	4.157	16.079	30.880	55.793
		(cov)	(0.365)	(0.244)	(0.217)	(0.132)	(0.137)
VAV92	$\delta = 10$	cpu sec	1.858	87.809	556.157	1713.139	*[1]
		(cov)	(0.006)	(0.009)	(0.007)	(0.004)	*
		rel. error	57.8%	40.9%	32.5%	48.8%	*
	$\delta = 100$	cpu sec	1.862	87.025	558.307	1711.397	*
		(cov)	(0.018)	(0.008)	(0.005)	(0.005)	*
		rel. error	92.1%	74.3%	65.8%	76.1%	*

1. 30 minutes cpu time limit or memory exceeded for all problems

Table 3.3.: Average CPU times for large problems using GN1$(n, \delta, 0, 0.5)$

Solver name	Data size parameter	Solution details	Problem size, n 30,000	50,000	70,000	100,000	150,000
OBG	$\delta = 10$	cpu sec	0.34	0.68	1.07	1.65	2.46
		(cov)	(0.471)	(0.491)	(0.399)	(0.653)	(0.392)
	$\delta = 100$	cpu sec	1.51	3.74	5.90	9.35	20.01
		(cov)	(0.071)	(0.206)	(0.309)	(0.376)	(0.242)
CPX-G	$\delta = 10$	cpu sec	64.84	215.47	547.40	1002.24	*[1]
		(cov)	(0.081)	(0.094)	(0.066)	(0.024)	*
	$\delta = 100$	cpu sec	142.19	700.60**	1044.88**[2]	*	*
		(cov)	(0.127)	(0.829)**	(0.262)**	*	*

1. 30 minutes cpu time limit or memory exceeded for all problems
2. 30 minutes cpu time limit or memory exceeded for at least one problem

and it is depicted in Figure 3.5.

As shown in Table 3.4, as the proportion of linear terms in the objective function increases, both OBG and CPX-G consumes less time; however, the Figure 3.5a shows that the decreasing rate of CPX-G is higher as size increases. The behavior of OBG and CPX-G in various proportion of concave terms is presented in Table 3.5 and Figure 3.5b. Although theoretically OBG has the worst time consuming case when $\nu = 0.5$, ironically it performs best in the experimented instances. On the other hand, CPX-G tends to consume more time as concave terms incase.

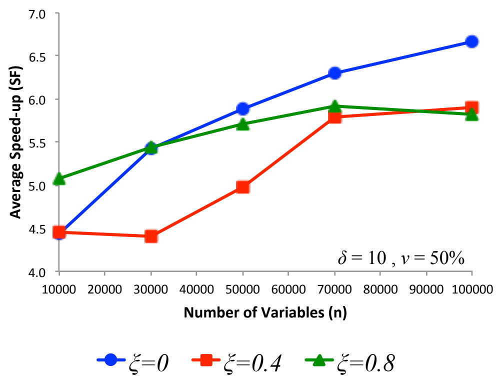
However, the speed of `OBG` is superior with high values of `SF`.

Table 3.4.: Sensitivity on ζ (average CPU times (sec.) for `GN1`$(n, 10, \zeta, 0.5)$)

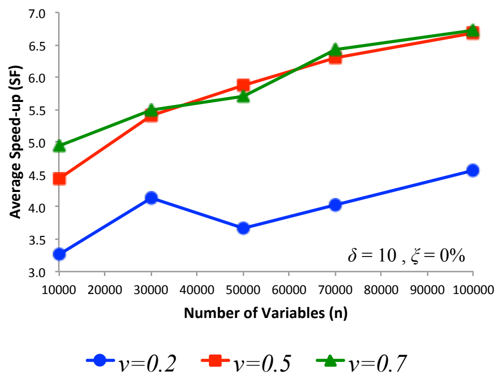
Solver	$\nu = 50\%$	Problem size, n				
name	ζ	10,000	30,000	50,000	70,000	100,000
`OBG`	0	0.09	0.34	0.68	1.07	1.65
	0.4	0.05	0.46	0.85	0.94	1.71
	0.8	0.02	0.07	0.12	0.24	0.55
`CPX-G`	0	7.30	64.84	215.47	547.40	1002.24
	0.4	4.33	34.27	115.61	269.71	523.46
	0.8	2.74	16.66	33.54	80.85	182.71

Table 3.5.: Sensitivity on ν (average CPU times (sec.) for `GN1`$(n, 10, 0, \nu)$)

Solver	$\zeta = 0\%$	Problem size, n				
name	ν	10,000	30,000	50,000	70,000	100,000
`OBG`	0.2	0.2	0.5	2.0	3.3	3.8
	0.5	0.1	0.3	0.7	1.1	1.7
	0.7	0.1	0.4	1.0	1.4	2.5
`CPX-G`	0.2	4.2	26.8	73.8	169.2	346.2
	0.5	7.3	64.8	215.5	547.4	1002.2
	0.7	11.2	93.9	298.5	713.4	1738.7



(a) Average speed-up of `OBG` relative to `CPX-G` on n and ζ



(b) Average speed-up of `OBG` relative to `CPX-G` on n and ν

Figure 3.5.: Average speed-up factor

Sensitivity of `OBG` solver in million sizes

The sensitivity to ν and ζ (the proportions of negative and zero d_j) of `OBG` is measured in very large sizes (n from 0.5 to 2 millions). Since `CPX-G` consumes over one hour for 0.5 millions size, only `OBG` is considered for this experiment. The sensitivity to the parameters (δ, ζ, and ν) is measured by comparing average speeds, `a` and `b`, at two levels of parameters with the relative speed,

$$Relative\ Speed := \frac{max\{avg - CPU - time(\mathrm{a}), avg - CPU - time(\mathrm{b})\}}{min\{avg - CPU - time(\mathrm{a}), avg - CPU - time(\mathrm{b})\}},$$

which is designed to be the value ≥ 1. As the metric is closes to 1, the average CPU times in two parameter levels `a` and `b` are insensitive. Parameter combinations of $\zeta = 0,\ 20\%$, $\nu = 0,\ 50\%$, and $\delta = 10,\ 100$ are used to run 10 instances, and the relative speed is in Table 3.6. Average relative speeds for each combination are ranged from 1.14 to 1.80. Thus, `OBG` solver may be concluded to be fairly insensitive for proportions of concave or linear terms of the objective function even in the millions sizes.

Table 3.6.: Relative speed of `OBG` on (n, δ, ζ, ν)

Compared parameters	Parameters	Data size parameter	Problem size, n, in millions					Average
			0.5	0.75	1	1.25	1.5	
$\zeta = 0$ *vs.* 0.2	$\nu = 0.5$	$\delta = 10$	1.15	1.93	1.30	1.08	1.10	1.31
		$\delta = 100$	1.43	1.20	1.11	1.15	1.19	1.22
	$\nu = 0.8$	$\delta = 10$	1.27	1.00	1.88	1.26	2.19	1.52
		$\delta = 100$	1.48	1.37	1.70	1.28	1.38	1.44
$\nu = 0.5$ *vs.* 0.8	$\zeta = 0$	$\delta = 10$	1.03	1.03	1.34	1.25	1.04	1.14
		$\delta = 100$	1.08	1.12	1.47	1.11	1.16	1.19
	$\zeta = 0.8$	$\delta = 10$	1.42	2.00	1.83	1.70	2.06	1.80
		$\delta = 100$	1.96	1.84	1.29	1.62	1.90	1.72

1. CPX-G is not tested because it spends over 1 hour in the tested sizes.
2. CPU times for each instances are in Appendix A.7.1.
3. As the values close to 1, the corresponding instance is insensitive to the compared parameters.

3.4.3. Experiments on Bounding with `OBG`

The quality of bounding procedure considered in Section 3.3 is tested with two random problems: `GN2` and subset-sum problem. The results of subset-sum problem is described in

Section 5.2.2. GN2 is generated following the steps for GN1, but it omits step 5 that changes $u_j = +\infty$. Parameters for GN2 is given by $\delta = 10$, $\zeta = 0$, and $\nu = 0.5$, and 10 instances are run for each instance in sizes from 500 to 50,000. The quality is measured by relative error with the global objective value from CPX-G. An upper bounding solver VAV92 is also considered, and average values of all results are in Table 3.7.

Clearly, VAV92 generates better quality upper bounds than OBG-based UB; however, VAV92 is not viable for large sizes ($n \geq 30,000$) within 2 hour time limit, while the OBG-based upper bounding procedure finds UB in a reasonable time.

Table 3.7.: Quality and speed of upper bounds on closed-box problems with GN2$(n, 10, 0, 0.5)$

	Problem size (n)	500	1,000	2,500	3,500	5,000
UB	rel. error[1]	5.62%	6.07%	5.19%	5.36%	5.05%
	cpu sec.	0.184	0.480	2.335	4.545	8.695
VAV92	rel. error[2]	1.22%	1.21%	1.25%	1.19%	1.19%
	cpu sec.	0.646	2.834	23.278	52.499	129.266

	Problem size (n)	10,000	20,000	30,000	40,000	50,000
UB	rel. error	5.51%	4.84%	5.27%	5.12%	5.09%
	cpu sec.	34.1	160.7	338.5	583.5	890.5
VAV92	rel. error	1.24%	1.21%	*	*	*
	cpu sec.	815.5	5656.2	*[3]	*	*

1. Relative error is defined by [UB - obj(CPX-G)]/abs[obj(CPX-G)]
2. Relative error is defined by [obj(VAV92) - obj(CPX-G)]/abs[obj(CPX-G)]
3. Two hour cpu time limit or memory exceeded for all problems

3.5. Conclusion

We have presented a quadratic time algorithm OBG that finds a global optimum for a class of indefinite knapsack separable quadratic programs with open box constraints. The two-stage decomposition approach leads necessary conditions, and our algorithm enumerate all KKT points that satisfy the necessary conditions efficiently employing the interval test technique. Moreover, based on the five global cases, the efficiency of algorithm is enhanced by detecting the global optimality with the least operations, and the worst case complexity of $O(n^2)$ is confirmed. Computational experiments turn out that OBG is quite insensitivity to the proportion of the negative and zero d_j, and it is superior in speed comparing to all other tested solvers.

The bounding algorithm that finds lower and upper bounds based on `OBG` is also developed to solve the indefinite knapsack separable quadratic programs with closed box constraints, which is already known as an *NP-hard* problem. Although the quality of the proposed algorithm is not preeminent to other solvers, it generates a quite acceptable feasible upper bound solution in the reasonable time limit.

4. Lagrangian-relaxed Closed-Box Solution

4.1. Introduction

In this chapter, another new methodology is developed to solve the indefinite case of (P) in (2.2) using a different approach to that developed in Chapter 3. In this approach, the knapsack constraint $\sum_{\forall j} x_j = b$ is relaxed in Lagrangian fashion. Toward this, we define the dual problem (D):

$$(D)\ Max_{\lambda \in \mathbb{R}}\ R(\lambda)$$

where $\lambda \in \mathcal{R}$ and $R(\lambda)$ is given by

$$\begin{aligned} R(\lambda) := \quad Min \quad & \tfrac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{c}^T\mathbf{x} + \lambda(\textstyle\sum_{\forall j} x_j - b) \\ s.t. \quad & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}. \end{aligned} \tag{4.1}$$

This relaxation approach in (4.1) is quite attractive because the problem is completely separable in all of the problem variables. Thus, an optimal solution x_j in $R(\lambda)$ is easily obtainable for each x_j for a given λ, and consequently, the dual problem (D) can be very efficiently solved by searching over the univariate λ.

Since the optimum solution (D) guarantees a lower bound of (P) by the weak duality theorem, see [10], if the optimum solution of (D) is feasible to (P), it is also a global optimum solution. Otherwise, the solution guarantees only a lower bound. In the latter case, we develop a new method that finds an upper bounding solution, which is feasible in (P), based on the lower bound solution. Therefore, this chapter present an algorithm, named Closed-box solver (`CBS`), that attempt KSQP solution in two stages:

Stage-1: Solve (D) to get a lower bound solution $\mathbf{x}(\lambda^*)$

Stage-2: If $\mathbf{x}(\lambda^*)$ is not feasible to (P), get a feasible upper bound solution $\mathbf{x}^{U*}$

Although the upper bound solution $\mathbf{x}^{U*}$ has no guarantee of being either a local or a global optimum, the gap between the lower and upper bounds is expected to be tight because the gap is determined by given coefficients of a single variable. Consequently, our experiments show that the quality of $\mathbf{x}^{U*}$ is significantly better than the local solution found by the local solver in [132].

The algorithm `CBS` is proposed in section 4.2, and section 4.3 presents the our new hybrid methods that combine the techniques used for strictly convex KSQP. Through the extensive experiments in section 4.4, the performance of our hybrid methods are compared with the existing pure methods to pick the best hybrid method, and it is compared with the local method [132], global method [23], and commercial solvers, Cplex and Matlab.

4.2. Closed-Box Solver (`CBS`)

This section derives solution characteristics to generate a lower bound solution, which is a global optimum solution of $R(\lambda)$, via geometric intuition. Then, a linear time procedure is proposed to construct a feasible upper bound solution based on the lower bound solution. The tightness of lower and upper bounds are discussed, and the cases that `CBS` finds an exact global optimum are explained.

4.2.1. Dual problem

Solution characteristics of dual function $R(\lambda)$

The global solution of $R(\lambda)$ in (4.1) for a given λ is defined by the following characteristics:

$$x_j(\lambda) = \begin{cases} median\{l_j, u_j, -\lambda/d_j\} & for\ j \in P \\ \begin{cases} l_j & if\ \lambda > c_j \\ u_j & otherwise \end{cases} & for\ j \in Z \\ \begin{cases} l_j & if\ \lambda > -d_j(l_j+u_j)/2 \\ u_j & otherwise \end{cases} & for\ j \in N \end{cases} \tag{4.2}$$

To show the above characteristics, define

$$q_j(x_j, \lambda) = \frac{1}{2}d_j x_j^2 - c_j x_j + \lambda x_j.$$

Then,

$$R(\lambda) = \sum_{\forall j} q_j(x_j(\lambda), \lambda).$$

Because $R(\lambda)$ is the sum of separate linear or quadratic functions with only box constraints $l_j \le x_j \le u_j$, the solution is quite straitforward for each variable sets.

- For $j \in P$, since $q_j(x_j, \lambda)$ is convex, it is minimized at $\frac{d}{dx_j}q_j(x_j, \lambda) = (d_j x_j + \lambda) = 0$ for a given λ and if $x_j = -\lambda/d_j$ is within the box constraint $l_j \le -\lambda/d_j \le u_j$. If $x_j = -\lambda/d_j$ is out of the box constraint, a closer extreme point l_j or u_j results in the minimum value of $q_j(x_j, \lambda)$. It implies

$$\begin{aligned} x_j &= \begin{cases} l_j & if\ \lambda \ge -d_j l_j \\ u_j & if\ \lambda \le -d_j u_j \\ -\lambda/d_j & otherwise \end{cases} \\ \Rightarrow x_j(\lambda) &= median\{l_j, u_j, -\lambda/d_j\} \end{aligned} \tag{4.3}$$

The same result can be derived by KKT conditions as in (2.5) or in [50].

- For $j \in Z$, the the optimum solution of the objective function $q_j(x_j, \lambda) = (-c_j + \lambda)x_j$ for a given λ can be determined by the sign of $(-c_j + \lambda)$. Moreover, if $\lambda = c_j$, the objective value $q_j(x_j, \lambda)$ is *zero* for any x_j, that is

$$q_j(x_j, c_j) = 0 \quad for\ x_j \in [l_j, u_j] \tag{4.4}$$

so we can choose any arbitrary value within $[l_j, u_j]$ in that case and set it to u_j in our algorithm. It implies

$$x_j = \begin{cases} l_j & -c_j + \lambda > 0 \\ u_j & -c_j + \lambda < 0 \\ [l_j, u_j] & -c_j + \lambda = 0 \end{cases} \tag{4.5}$$

$$\Rightarrow x_j(\lambda) = \begin{cases} l_j & if\ \lambda > c_j \\ u_j & otherwise \end{cases} \tag{4.6}$$

- For $j \in N$, the maximum value of $q_j(x_j, \lambda)$ is at $x_j = -\lambda/d_j$ for a given λ, and the minimum value can be obtained at l_j (or u_j), if $x_j = -\lambda/d_j$ is farther from l_j (or u_j) than u_j (or l_j). Moreover, if $x_j = -\lambda/d_j$ is the middle point of $[l_j, u_j]$, that is $x_j = (l_j + u_j)/2$ yielding $\lambda = -d_j(l_j + u_j)/2$, then $q_j(x_j, \lambda)$ is minimized at either $x_j = l_j$ or u_j having the same value of $-d_j l_j u_j/2$, that is,

$$q_j(l_j, t_j) = q_j(u_j, t_j) = -d_j l_j u_j/2 \quad for\ t_j = -d_j(l_j + u_j)/2 \tag{4.7}$$

So we set it to an arbitrary value u_j as we do for $j \in Z$. It implies

$$\begin{aligned} x_j &= \begin{cases} l_j & if\ \frac{-\lambda}{d_j} > \frac{l_j+u_j}{2} \\ u_j & if\ \frac{-\lambda}{d_j} < \frac{l_j+u_j}{2} \\ l_j\ or\ u_j & if\ \frac{-\lambda}{d_j} = \frac{l_j+u_j}{2} \end{cases} \\ \Rightarrow x_j(\lambda) &= \begin{cases} l_j & if\ \lambda > -d_j(l_j+u_j)/2 \\ u_j & otherwise. \end{cases} \end{aligned} \tag{4.8}$$

See Appendix A.4 for another derivation for (4.8).

Note that $\mathbf{x}(\lambda)$ satisfies all KKT conditions in (2.4) of (P) except for the knapsack constraint $\sum_{\forall j} x_j = b$.

Definition 4.1. *Breakpoint*

We define that breakpoints are the values on λ which make $x_j(\lambda)$ be non-differentiable. $x_j(\lambda)$ for $j \in P$ generates two beak points

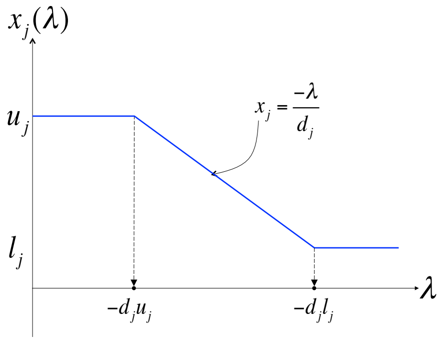
$$-d_j u_j, \text{ and } -d_j l_j$$

and each variable for others sets generate one breakpoint

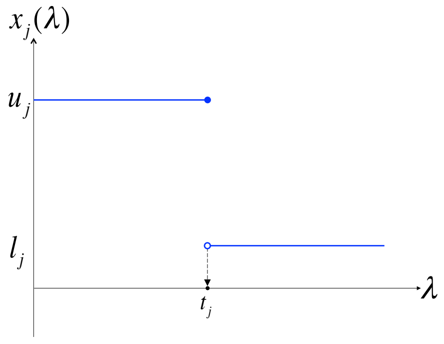
$$t_j = \begin{cases} c_j & for\ j \in Z \\ -d_j(l_j+u_j)/2 & for\ j \in N \end{cases} \tag{4.9}$$

Note that the value of t_j is same for original KSQP and transformed (P) because transformation (2.3) is linear and separate.

Theorem 4.2. *$x_j(\lambda)$ is a non-increasing piecewise linear function on λ and discontinuous at t_j.*



(a) $x_j(\lambda)$ for $j \in P$



(b) $x_j(\lambda)$ for $j \in Z \cup N$

Figure 4.1.: $x_j(\lambda)$ is a non-increasing piecewise linear function on λ

Proof. For the case of $j \in P$, as shown in Figure 4.1a, it is easy to see that $x_j(\lambda)$ is a constant value of u_j for λ less than lower breakpoint $-d_j u_j$ and is continued as a decreasing linear function with the slope of $-1/d_j$ until the value is reached to l_j at the upper breakpoint $-d_j l_j$. The line is also continued with a constant value of l_j for λ greater than the upper breakpoint. For other cases of $j \in Z \cup N$, we can also easily see that $x_j(\lambda)$ is u_j for λ less than or equal to breakpoint t_j and is l_j otherwise. The Figure 4.1b shows it with a discontinuous point at a breakpoints t_j. □

Theorem 4.3. *$R(\lambda)$ is a continuous piecewise concave function and non-differentiable at breakpoints t_j for $j \in Z \cup N$*

Proof. The theorem can be proved by each feature.

Continuity and Concavity (page 215, Bazaraa et al. [10])

Let $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{c}^T\mathbf{x}$, $g(\mathbf{x}) = \mathbf{a}^T\mathbf{x} - b$, and $B = \{\mathbf{x} : \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$. Since $\lambda \in \mathbb{R}$ and B is a nonempty compact set in $\mathbb{R}^n$, $R(\lambda)$ is continuous and finite everywhere on $\lambda \in \mathbb{R}$. Let

$\alpha \in (0,1)$. By the definition of concave function, we have

$$\begin{aligned}
R(\alpha\lambda_1 + (1-\alpha)\lambda_2) &= Min\{f(\mathbf{x}) + [\alpha\lambda_1 + (1-\alpha)\lambda_2]g(\mathbf{x}) : \mathbf{x} \in B\} \\
&= Min\{\alpha[f(\mathbf{x}) + \lambda_1 g(\mathbf{x})] + (1-\alpha)[f(\mathbf{x}) + \lambda_2 g(\mathbf{x})] : \mathbf{x} \in B\} \\
&\geq \alpha Min\{f(\mathbf{x}) + \lambda_1 g(\mathbf{x}) : \mathbf{x} \in B\} + (1-\alpha) Min\{f(\mathbf{x}) + \lambda_2 g(\mathbf{x}) : \mathbf{x} \in B\} \\
&= \alpha R(\lambda_1) + (1-\alpha)R(\lambda_2)
\end{aligned}$$

Thus, $R(\lambda)$ is a continuous concave function.

Piecewise function and non-differentiability

We define $R_j(\lambda) = \frac{1}{2}d_j x_j^2(\lambda) - c_j x_j(\lambda) + \lambda x_j(\lambda)$ be a separate part of $R(\lambda)$ for each j. That is, $R(\lambda) = \sum_{\forall j} R_j(\lambda)$. Then, we can see that $R_j(\lambda)$ is a linear or quadratic function on λ in (4.10), (4.11), and (4.12), and examples are illustrated in Figure 4.2a, 4.2b, and 4.2c.

• For $j \in P$, we can show that $R_j(\lambda)$ is a continuous piecewise concave function substituting $x_j(\lambda)$ in (4.3) into $R_j(\lambda)$ as (4.10). $R_j(\lambda)$ is a quadratic concave function if λ is between breakpoints ($\lambda \in [-d_j u_j,\ -d_j l_j]$) and two linear lines are continued with the slope at each breakpoint. So $R_j(\lambda)$ is continuous and differentiable everywhere.

$$R_j(\lambda) = \frac{1}{2}d_j x_j^2(\lambda) + \lambda x_j = \begin{cases} d_j l_j^2/2 + \lambda l_j & if\ \lambda \geq -d_j l_j \Rightarrow Linear\ function \\ d_j u_j^2/2 + \lambda u_j & if\ \lambda \leq -d_j u_j \Rightarrow Linear\ function \\ -\lambda^2/(2d_j) & otherwise \Rightarrow Quadratic\ function \end{cases} \tag{4.10}$$

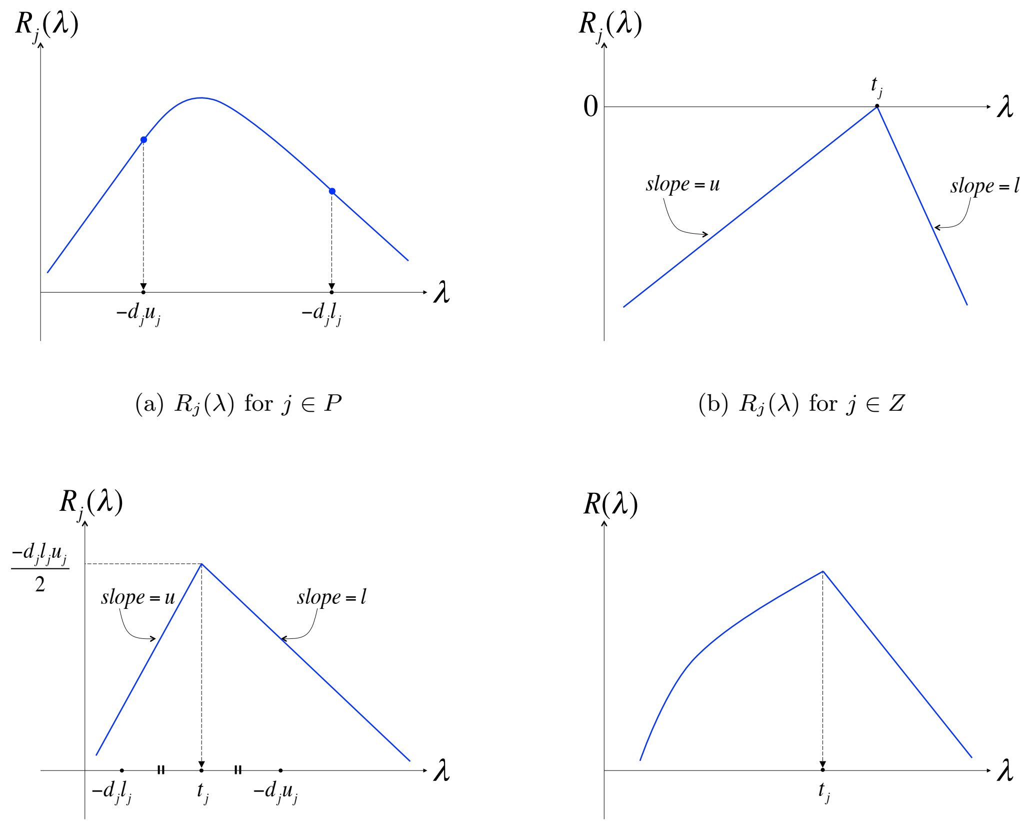
• For $j \in Z$, the objective function $R_j(\lambda) = (-c_j + \lambda)x_j(\lambda)$ consists of two linear lines with the slope of u_j and l_j as in (4.11). Thus, it is a continuous piecewise linear concave function and is non-differentiable at a break point t_j. Note that the objective value at the breakpoint is $R_j(t_j) = 0$ for any x_j as (4.4).

$$R_j(\lambda) = (-c_j + \lambda)x_j = \begin{cases} -c_j l_j + \lambda l_j & if\ \lambda > c_j \Rightarrow Linear\ function \\ -c_j u_j + \lambda u_j & otherwise \Rightarrow Linear\ function \end{cases} \tag{4.11}$$

- For $j \in N$, $R_j(\lambda)$ is like the case of $j \in Z$ as in (4.12), but $R_j(t_j)$ has the same value of $-d_j l_j u_j/2$ for either $x_j = l_j$ or u_j as noticed in (4.7).

$$R_j(\lambda) = \frac{1}{2}dx_j^2 + \lambda x_j = \begin{cases} d_j l_j^2/2 + \lambda l_j & if\ \lambda > -d_j(l_j+u_j)/2 \Rightarrow Linear\ function \\ d_j u_j^2/2 + \lambda u_j & otherwise \Rightarrow Linear\ function \end{cases} \tag{4.12}$$

Since $R(\lambda)$ is the sum of $R_j(\lambda)$ that are piecewise concave and non-differentiable at breakpoints t_j, $R(\lambda)$ also has the same properties (see an example in Figure 4.2d). Therefore, the theorem holds. $\square$



(a) $R_j(\lambda)$ for $j \in P$

(b) $R_j(\lambda)$ for $j \in Z$

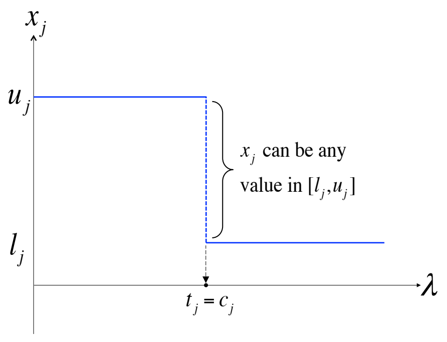
(c) $R_j(\lambda)$ for $j \in N$

(d) $R(\lambda)$

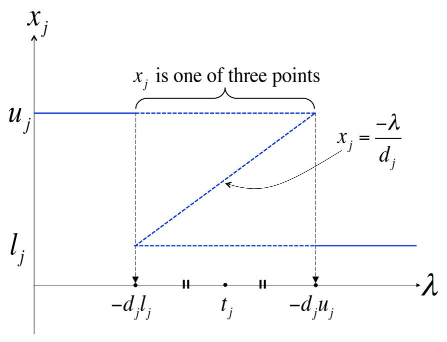
Figure 4.2.: $R(\lambda) = \sum_{\forall j} R_j(\lambda)$ is a continuous piecewise concave function

Duality Gap

If we let f^* be the global optimum solution of (P), by the weak duality theorem (see [10]), it always holds that $R(\lambda^*) \leq f^*$ for $\lambda \in \mathcal{R}$. If $R(\lambda^*) < f^*$, the difference $f(x^*) - R(\lambda^*)$ is called *duality gap*. In 4.6 and 4.8, we set $x_j(t_j)$ to be u_j for $j \in Z \cup N$; however, the KKT conditions of (P) in (2.4) implies that at $\lambda = t_j$, x_j can be a value as $x_j \in [l_j, u_j]$ for $j \in Z$ and $x_j \in \{l_j, u_j, -\lambda/d_j\}$ for $j \in N$. It is drawn as in Figure 4.3. Then, one can easily see the difference between the Figure 4.3 and Figure 4.1b for $x_j(\lambda)$ $j \in Z \cup N$. Therefore, even in the case that λ^* of (D) is the global optimum value in KKT conditions of (P), the *duality gap* can occur due to the way we set $x_j(t_j) = u_j$ for $j \in Z \cup N$.



(a) x_j for $j \in Z$



(b) x_j for $j \in N$

Figure 4.3.: x_j for $j \in Z$ and $j \in N$

Domain of Lagrange multiplier λ

The domain of λ is unrestricted in (D) , but from $\mathbf{x}(\lambda)$ we can observe that breakpoints can be used to restrict the range of λ. Here we show that the observation can be proved by KKT conditions (2.4). The dual domain varies dependent on the bounds in the box constraints $(\mathbf{l} \leq \mathbf{x} \leq \mathbf{u})$. If all bounds for x_j are finite, we can get the minimum domain of λ simply from break points. However, if x_j are one sided bounded for some $j \in Z$, that is, $l_j \leq x_j$, we can further narrower the domain using KKT conditions.

Lemma 4.4. *If l_j and u_j are finite values for $j \in Z$, then $\lambda^* \in [\lambda_{min},\ \lambda_{max}]$ where*

$$\begin{aligned}\lambda_{min} &= min\{-d_i u_i,\ t_j : i \in P,\ j \in Z \cup N\} \\ \lambda_{max} &= max\{-d_i l_i,\ t_j : i \in P,\ j \in Z \cup N\}\end{aligned}$$

are minimum and maximum break points.

Proof. If (P) is bounded, b must be bounded as

$$\sum_{\forall j} l_j \leq b \leq \sum_{\forall j} u_j,$$

and each side of bounds can be obtained by $\mathbf{x}(\lambda)$ at λ_{min} and $\lambda_{max} + \varepsilon$, $\varepsilon > 0$ because

$$x_j(\lambda_{min}) = u_j \quad \text{and} \quad x_j(\lambda_{max} + \varepsilon) = l_j \quad \forall j$$

Notice that ε can be ignored and we can argue λ^* is in the range $[\lambda_{min},\ \lambda_{max}]$ because a part objective value $R_j(\lambda)$ has the same value at either $x_j = l_j$ or u_j for $t_j = \lambda_{max}$. In other words,

$$R_j(\lambda_{max}) = R_j(\lambda_{max} + \varepsilon)$$

having values of

$$\begin{aligned} R_j(\lambda_{max}) = &\ 0 && for\ x_j \in [l_j, u_j],\ t_j = \lambda_{max},\ j \in Z \\ R_j(\lambda_{max}) = &\ -d_l l_j u_j/2 && for\ x_j \in \{l_j, u_j\},\ t_j = \lambda_{max},\ j \in N \end{aligned} \tag{4.13}$$

as derived in (4.6) and (4.8). □

Lemma 4.5. *If l_j and u_j are infinite values for some $j \in Z$, then $\lambda^* \in [\lambda_{min},\ \lambda_{max}]$ where*

$$\begin{aligned}\lambda_{min} &= max\{\hat{\lambda}_{min},\ t_j : j \in Z_u\} \\ \lambda_{max} &= min\{\hat{\lambda}_{max},\ t_j : j \in Z_l\}\end{aligned}$$

and

$$\begin{aligned} \hat{\lambda}_{min} &= min\{-d_i u_i,\ t_j : i \in P,\ j \in (Z \cup N) \backslash (Z_u \cup Z_l)\} \\ \hat{\lambda}_{max} &= max\{-d_i l_i,\ t_j : i \in P,\ j \in (Z \cup N) \backslash (Z_u \cup Z_l)\} \end{aligned} \tag{4.14}$$

for $Z_l = \{j \in Z : l_j = -\infty\}$*, and* $Z_u = \{j \in Z : u_j = \infty\}$*.*

Proof. $\hat{\lambda}_{min}$ and $\hat{\lambda}_{max}$ are proved in Lemma 4.4. Consider other parts with KKT conditions.

If $l_j = -\infty$ for some $j \in Z$ by the stationarity KKT conditions (2.4),

$$\begin{aligned} -c_j + \lambda + \mu_j &= 0 \\ -c_j + \lambda &= -\mu_j \leq 0 \\ \lambda &\leq c_j = t_j \end{aligned}$$

Thus, $\lambda^* \leq \lambda_{max} \leq t_j$ for $j \in Z_l$.

If $u_j = \infty$ for some $j \in Z$ by the stationarity KKT conditions (2.4)

$$\begin{aligned} -c_j + \lambda - \gamma_j &= 0 \\ -c_j + \lambda &= \gamma_j \geq 0 \\ \lambda &\geq c_j = t_j \end{aligned}$$

Thus, $\lambda^* \geq \lambda_{min} \geq t_j$ for $j \in Z_u$, and Lemma is complete. □

A similar result of Lemma 4.5 is presented in Lotito (2006, proposition 3 of [74]) with one sided bound $0 \leq x_j$ for positive semidefinite case ($N = \emptyset$), but the proof does not rely on KKT conditions. Note that the one bounded cases for $j \in N$ was considered in the previous Section 3.2.

If (P) has the inequality knapsack constraint $\sum_{\forall j} x_j \leq b$, then it is trivial that $\lambda^* \geq 0$ from KKT conditions (2.4). This case has been considered in the literature such as Stefanov [127] and Bretthauer et al. [18]. Practically, in the inequality case, we can first test whether the knapsack constraint is satisfied at $\lambda = 0$ by

$$\sum_{\forall j} x_j(0) \leq b.$$

If the inequality is satisfied, $\mathbf{x}(0)$ is a global optimum of (P). If it is not satisfied, simply we can search for a $\lambda^* > 0$ within $[\lambda_{min}, \lambda_{max}]$ as defined in Lemma 4.4 and 4.5.

Theorem 4.6. *$R(\lambda)$ is maximized at a $\lambda \in [\lambda_{min},\ \lambda_{max}]$*

Proof. Since $R(\lambda)$ is a concave function as proved by Theorem 4.3 and $\mathbf{x}(\lambda)$ covers all feasible solutions of $R(\lambda)$ with $\lambda \in [\lambda_{min}, \lambda_{max}]$ by Lemma 4.4 and 4.5, the theorem holds. Hence, (D) has an optimum solution at the $\lambda^* \in [\lambda_{min},\ \lambda_{max}]$. □

4.2.2. Solving the dual problem

Theorem 4.3 and Theorem 4.6 imply that $R(\lambda)$ is concave having the maximum point in the range of $\lambda \in [\lambda_{min},\ \lambda_{max}]$. That means λ^* is a value that satisfies

$$\frac{d}{d\lambda}R(\lambda^* - \varepsilon) \geq 0 \geq \frac{d}{d\lambda}R(\lambda^* + \varepsilon)$$

for a given $\varepsilon \geq 0$ such that $\lambda^* - \varepsilon \neq t_j$, and $\lambda^* + \varepsilon \neq t_j$ for $j \in Z \cup N$.

Right and Left derivatives of $R(\lambda)$

$R(\lambda)$ can be non-differentiable at λ^* if and only if $\lambda^* = t_j$ for some $j \in Z \cup N$ as proved in Theorem 4.3. Figure 4.2 illustrates an example. So we may search for λ^* such that the right and left derivatives of $R(\lambda)$ have opposite signs or at least one is *zero*.

For $\varepsilon \geq 0$, consider the following two derivatives:

$$\begin{aligned} \textit{Right derivative}: \quad g(\lambda) &= \lim_{\varepsilon \to 0}[R(\lambda) - R(\lambda - \varepsilon)]/\varepsilon \\ &= \sum_{\forall j} x_j(\lambda) - b \end{aligned}$$

$$\begin{aligned} Left\ derivative: \quad g_l(\lambda) &= \lim_{\varepsilon\to 0}[R(\lambda) - R(\lambda+\varepsilon)]/\varepsilon \\ &= \sum_{\forall j} \hat{x}_j(\lambda) - b \\ &= g(\lambda) + \sum_{j\in K(\lambda)} (l_j - u_j) \end{aligned}$$

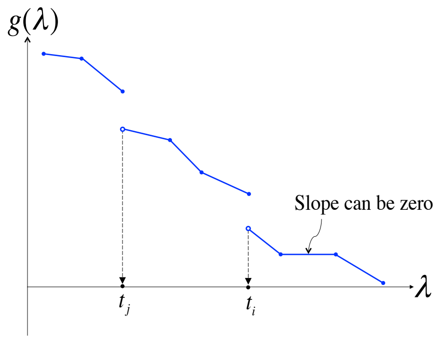
where

$$K(\lambda) = \{k \in Z \cup N : t_k = \lambda\},$$

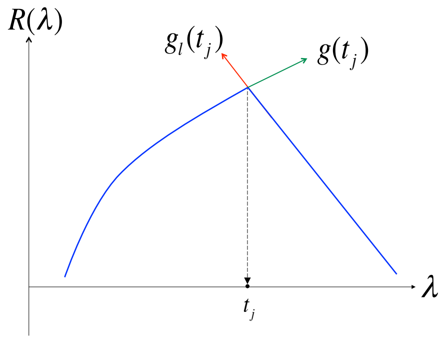
and $\hat{\mathbf{x}}(\lambda)$ is define by

$$\hat{x}_j(\lambda) = \begin{cases} l_j & for\ j \in K(\lambda) \\ x_j(\lambda) & for\ j \notin K(\lambda). \end{cases} \tag{4.15}$$

Since $x_j(\lambda)$ is a non-increasing piecewise linear function and discontinuous at t_j, the function $g(\lambda)$, which consists of a constant b and the sum of $x_j(\lambda)$, has the same properties. An example of $g(\lambda)$ is illustrated in Figure 4.4a, and an example of right and left derivatives is illustrated in Figure 4.4b.



(a) $g(\lambda)$ is a non-increasing piecewise linear function and discontinuous at t_j



(b) Right derivative $g(\lambda)$ and left derivative $g_l(\lambda)$ of $R(\lambda)$

Figure 4.4.: $g(\lambda)$ is the right derivative (slope) of $R(\lambda)$

Upper bound solution $\mathbf{x}^{U*}$

Note that both $g(\lambda)$ and $g_l(\lambda)$ represent the violation of knapsack constraint. If one of derivatives (violations) is *zero* at λ^*, then the corresponding solution $\mathbf{x}(\lambda^*)$ or $\hat{\mathbf{x}}(\lambda^*)$ is a global optimum solution because it is feasible in (P) and guarantees the lower bound of (P) by weak duality theorem. Otherwise, $\mathbf{x}(\lambda^*)$ and $\hat{\mathbf{x}}(\lambda^*)$ are infeasible in the knapsack constraint and guarantees only a lower bound. Note that the lower bound $R(\lambda)$ has the same value in either choice of $\mathbf{x}(\lambda^*)$ and $\hat{\mathbf{x}}(\lambda^*)$ due to the properties in (4.13).

However, a feasible solution $\mathbf{x}^{U*}$ can be found in the solution set X_U:

$$Upper\ bound\ solution\ set: \quad X_U = \left\{ \mathbf{x} : x_j = \begin{cases} \in [l_j, u_j] & for\ j \in K^* \\ x_j(\lambda^*) & for\ j \notin K^* \end{cases} \right\} \tag{4.16}$$

where

$$K^* = K(\lambda^*). \tag{4.17}$$

We name $\mathbf{x}^{U*}$ *upper bound solution* because it guarantees an upper bound of (P) as a feasible solution. Similarly, we also call $\mathbf{x}(\lambda^*)$ *lower bound solution*. Hence, the following inequalities hold

$$R(\lambda^*) \leq f(\mathbf{x}^*) \leq f(\mathbf{x}^{U*}),$$

where $f(\mathbf{x})$ is the objective function of (P). Moreover, if $K^* = \emptyset$, since $\mathbf{x}(\lambda^*)$ is a global optimum solution, we set $\mathbf{x}^{U*} = \mathbf{x}(\lambda^*)$, and no duality gap exists as

$$R(\lambda^*) = f(\mathbf{x}^*) = f(\mathbf{x}^{U*}).$$

The existence of $\mathbf{x}^{U*}$ is proved through the following Theorem 4.7.

Theorem 4.7. *If $K^* \neq \emptyset$, there exists an upper bound solution $\mathbf{x}^{U*} \in X_U$ that is feasible to (P).*

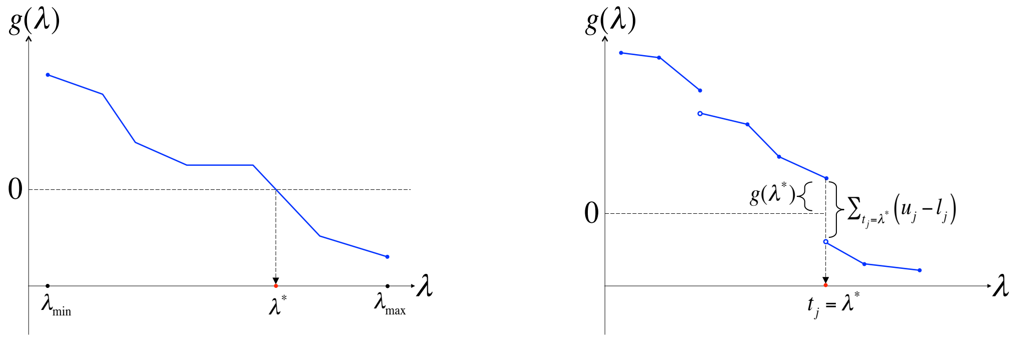
Proof. If $K^* \neq \emptyset$, then

$$\begin{aligned} g(\lambda^*) - g_l(\lambda^*) &= \sum_{j \in K} (u_j - l_j) \\ g(\lambda^*) &= g_l(\lambda^*) + \sum_{k \in K^*} (u_k - l_k) \quad \textit{since } g_l(\lambda^*) \leq 0 \\ &\leq \sum_{j \in K^*} (u_j - l_j) \end{aligned}$$

Moreover, because $l_j < u_j$, it holds that

$$\sum_{j \in K^*} (u_j - l_j) > 0$$

See Figure 4.5b for graphical intuition. Thus, $g(\lambda^*) \leq \sum_{k \in K^*} (u_k - l_k)$, and it implies that $\mathbf{x}^{U*}$ can be found by adjusting x_j in the range within $[l_j, u_j]$ for $j \in K^*$ and holding $x_j^{U*} = x_j(\lambda^*)$ for $j \notin K^*$. Hence, a feasible *upper bound* solution $\mathbf{x}^{U*} \in X_U$ exists. $\square$



(a) $g(\lambda)$ in a strictly convex case of (P) (b) $g(\lambda)$ in an non-strictly convex case of (P)

Figure 4.5.: $g(\lambda)$

Figure 4.5a is an example of $g(\lambda)$ for the strictly convex case of (P). It shows that $\mathbf{x}(\lambda^*)$ is always a global optimum for a feasible b since $g(\lambda^*)$ is continuous for $\lambda \in [\lambda_{min}, \lambda_{max}]$. On the other hand, a nonconvex case of (P) is drawn in Figure 4.5b with a discontinuous piecewise linear lines presenting Theorem 4.7.

Tightness of the upper and lower bounds; *bound gap*

There can exist difference between lower and upper bounds that are obtained through $\mathbf{x}(\lambda^*)$ and $\mathbf{x}^{U*}$. We name the gap *bound gap*:

$$Bound\ Gap = f(\mathbf{x}^{U*}) - R(\lambda^*).$$

The maximum *bound gap* is totally dependent on the given coefficients of a single variable for $j \in K^*$. Define the separate objective function $f_j(x_j) = \frac{1}{2}d_j x_j^2 - c_j x_j$. If only the box constraints are considered, the maximum value of $f_j(x_j)$ is

$$f_j = \begin{cases} \begin{cases} -c_j l_j & if\ c_j > 0 \\ -c_j u_j & if\ c_j < 0 \end{cases} & for\ j \in Z \cap K^* \\ \frac{1}{2}d_j \bar{x}_j^2 & for\ j \in N \cap K^* \end{cases}$$

where $\bar{x}_j = median\{l_j, 0, u_j\}$.

Since at most one x_j^* for $j \in N$ is strictly within (l_j, u_j) as will be proved in Theorem 4.8 in Section 4.2.3.3, if we find $\mathbf{x}^{U*}$ following the theorem, the maximum limit of *bound gap* is

$$Bound\ Gap \leq Max\left\{f_j - q_j(u_j, \lambda^*) : j \in K^*\right\}, \tag{4.18}$$

and it also gives the limit of *duality gap* by

$$Duality\ gap \leq Bound\ Gap$$

Note that we substitute u_j in $q_j(x_j, \lambda)$ for (4.18) because $q_j(l_j, \lambda^*) = q_j(u_j, \lambda^*)$ for $j \in K^*$, i.e.

$$q_j(x_j, \lambda^*) = \begin{cases} -d_j l_j u_j/2 & for\ j \in N \cap K^*\ and\ x_j = \{l_j, u_j\} \\ 0 & for\ j \in Z \cap K^*\ and\ x_j \in [l_j, u_j], \end{cases} \tag{4.19}$$

as shown in (4.4) and (4.7).

Finding an upper bound solution $\mathbf{x}^{U*}$; *weakness* of `CBS`

As we will discuss about the global solution in the next Section 4.2.3, we can find a $\mathbf{x}^{U*}$ as a global optimum solution if K^* is empty or includes at least one index in Z with some conditions in (4.24). However, if all indices in K^* are in N (i.e. $K^* \subseteq N$), finding the best $\mathbf{x}^{U*}$ is a strictly concave problem of $(P2)$

$$
\begin{aligned}
(P2) \quad Min \quad & \frac{1}{2} \sum_{j \in K^*} d_j x_j^2 \\
s.t. \quad & \sum_{j \in K^*} x_j = b - \sum_{j \notin K^*} x(\lambda^*) \\
& l_j \leq x_j \leq u_j \qquad for\ j \in K^*
\end{aligned}
$$

This strictly concave case is known to be *NP-hard* and has been considered by Moré and Vavasis (1991, [83]), but their algorithm gives only a local optimum in $O(n\ log\ n)$ time. This is the only *weakness* of `CBS` because even though we are able to find a global optimum of $(P2)$ taking exponential time with some methods such as branch and bound or dynamic programming, the feasible solution $\mathbf{x}^{U*}$ does not guarantee the global optimum to (P).

An extreme case is the subset-sum problem, which will be discussed in application Chapter 5.2.2, because all t_j for $j \in N$ are same as *zero* and the size of $K^* = N$ is $n-1$. However, the case that K^* includes multiple index may be practically rare because having the exactly same $t_j = -d_j(l_j + u_j)/2$ requires a special structure of (P) like the subset-sum problem. Furthermore, if there is only one index in $K^* \subseteq N$, there is a unique $\mathbf{x}^{U*}$, which can be found without computation by

$$
\mathbf{x}_j^{U*} = \begin{cases} g(\lambda^*) & j \in K^* \\ x_j(\lambda^*) & j \notin K^* \end{cases} \tag{4.20}
$$

4.2.3. Global solution

In this section, we show all cases that `CBS` can find a global optimum solution. To prove the global optimality, we present necessary conditions and sufficient conditions for optimality of

(P).

4.2.3.1. Necessary conditions

It is well known that KKT conditions are necessary conditions (see page 162 of [10]) to verify that a feasible solution is a local optimum solution. In addition to KKT conditions, Theorem 4.8 and 4.9 give more conditions for local optimality.

Theorem 4.8. *If $\mathbf{x}$ is a local optimum solution of (P), then x_j for $j \in N$ is at its extreme point l_j or u_j except for at most one $x_j^* \in (l_j, u_j)$.*

Proof. This is proved by theorem 2.2 and 2.3 of Moré and Vavasis [83] and by Lemma 4 of Vavasis [132], so we show a brief idea based on Lemma 4 of Vavasis [132].

By contradiction, suppose $\mathbf{x}$ is a local optimum solution and its two values are non-extreme value as $x_j \in (l_j, u_j)$ for $j = 1, 2$ and $j \in N$. Then, we can generate two feasible solutions

$$\begin{aligned}\mathbf{x}_1 &= (x_1 + \varepsilon, x_2 - \varepsilon, x_3, ...x_n)\\ \mathbf{x}_2 &= (x_1 - \varepsilon, x_2 + \varepsilon, x_3, ...x_n)\end{aligned}$$

for $\varepsilon > 0$. The average of two objective values $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$ is strictly smaller than $f(\mathbf{x})$. That is,

$$f(\mathbf{x}) + (d_1 + d_2)\varepsilon < f(\mathbf{x})$$

since $(d_1 + d_2) < 0$. This contradicts to a hypothesis that $\mathbf{x}$ is a local optimum with two non-extreme values for $j \in N$. □

Theorem 4.9. *A solution $\mathbf{x}$ is a local optimum solution of (P) if and only if it satisfies (4.21) or (4.22).*

If x_j for $j \in N$ is at its extreme point l_j or u_j

$$\begin{aligned}&-d_i l_i \le \lambda^* \le -d_j u_j \quad \textit{for } i \in L, j \in U\\ &-d_i l_i < -d_j u_j \qquad\quad \textit{for } i \in L, j \in U\end{aligned} \tag{4.21}$$

If x_j for $j \in N$ is at its extreme point l_j or u_j except for one $x_m \in (l_j, u_u)$ for $m \in N$

$$\begin{aligned} &-d_i l_i < \lambda^* < -d_j u_j \quad \textit{for } i \in L, j \in U \\ &\lambda^* = -x_m d_m \end{aligned} \tag{4.22}$$

for $L = \{i \in N : x_i = l_i\}$ *and* $U = \{j \in N : x_j = u_j\}$

Proof. This is suggested and proved in Lemma 5 and 6 of Vavasis [132]. It is basically strengthened KKT conditions to get sufficiency of local optimum based on $-d_j l_j$ and $-d_j u_j$, which are the negative value of derivative of $f_j(x_j) = \frac{1}{2} d_j x_j^2$ at its bounds. We omit the proof because this is well proved in [132], and above all, this cannot be used to determine global optimality in CBS. This theorem is in Section 2.4.1 to explain [83, 132]. □

4.2.3.2. Sufficient conditions

We have seen that $\mathbf{x}^{U*}$ is a global optimum if $K^* = \emptyset$. In addition to that, a sufficient condition that ensures the global optimality of $\mathbf{x}^{U*}$ is presented in Theorem 4.10 introducing a restricted solution $\mathbf{x}^R \in X_U$.

Theorem 4.10. *A solution* $\mathbf{x}^R \in X_U$ *is a global optimum solution, if it is feasible in* (P) *and restricted as*

$$x_j^R = \begin{cases} x_j(\lambda^*) & \textit{for } j \notin K^* \\ l_j \textit{ or } u_j & \textit{for } j \in K^* \cap N \\ [l_j, u_j] & \textit{for } j \in K^* \cap Z \end{cases} \tag{4.23}$$

Proof. It is possible that a feasible solution $\mathbf{x}^R$ exists in X_U due to Theorem 4.7. For $\mathbf{x}^R$,

$$q_j(x_j^R, \lambda^*) = q_j(x_j(\lambda^*), \lambda^*)$$

because $q_j(x_j^R, \lambda^*)$ stays at the same value for any x_j^R as shown in (4.19). Moreover, by the weak duality theorem, it is guaranteed that

$$f(\mathbf{x}^*) \geq R(\lambda^*) = \sum_{\forall j} q_j(x_j(\lambda^*), \lambda^*).$$

Therefore, if $\mathbf{x}^R$ is feasible in (P), it is a global optimum solution since

$$f(\mathbf{x}^*) = R(\lambda^*) = f(\mathbf{x}^R).$$

$\square$

4.2.3.3. Global optimum cases

This section presents four cases that $\mathbf{x}^{U*}$ can be a global optimum solution. Case 1 is for the convex case of (P) and other three cases are for indefinite cases of (P).

Case 1. $N = \emptyset$

This is the convex case of (P). So there exists a global optimum solution $\mathbf{x}^{U*}$ by strong duality theorem. Consider two cases by the emptiness of Z.

- Case 1.1 If $N = \emptyset$ and $Z = \emptyset$, a global optimum solution is $\mathbf{x}^{U*} = \mathbf{x}(\lambda^*)$.
- Case 1.2 If $N = \emptyset$ and $K^* \cap Z \neq \emptyset$, any $\mathbf{x}^{U*}$ that satisfies the knapsack constraint $(\sum_{\forall j} x_j = b)$, that is

  $$\mathbf{x}^{U*} = \{\mathbf{x} \in X_U : \sum_{\forall j} x_j = b\},$$

  is a global optimum solution. We can easily find a $\mathbf{x}^{U*}$ in $O(n)$ time distributing the value of $g(\lambda^*)$ to x_j for $j \in K^* \cap Z$ by any preferable ways. An easy way is suggested in Algorithm 4.1.

Case 2. $K^* = \emptyset$ **and** $Z \cup N \neq \emptyset$

This is the case that $g(\lambda)$ is continuous at its root as Figure 4.5a so $R(\lambda)$ is differentiable at λ^* and knapsack constraint $(\sum_{\forall j} x_j = b)$ is satisfied with $\mathbf{x}(\lambda^*)$. Therefore, a global optimum solution is $\mathbf{x}^{U*} = \mathbf{x}(\lambda^*)$.

Algorithm 4.1 $O(n)$; Find a global optimum solution $\mathbf{x}^{U*}$ for the case of $K^* \subseteq Z$

```
1. Find $K^Z = K^* \cap Z$
2. Set $\mathbf{x}^{U*} = \mathbf{x}(\lambda^*)$, $g_u = g(\lambda^*)$, $s = \sum_{j \in K^Z} u_j$, and $i = 1$
3. Iterate to distribute $g_u$ to $x_j \in [l_j, u_j]$ for $j \in K^Z$
    a) Set $j = K_i^Z$, $x_j^{U*} = max(l_j, u_j - g_u)$, and $g_u = g_u - u_j + x_j^{U*}$
    b) If $g_u = 0$, finish algorithm with $\mathbf{x}^{U*}$
       Else $i = i + 1$
```

Case 3 $K^* \cap Z \neq \emptyset$**,** $K^* \cap N \neq \emptyset$**, and** $(P3)$ **is feasible**

$$\begin{aligned} (P3) \quad & b_l \leq \sum_{j \in K^N} x_j \leq b_u \\ & x_j = \{l_j, u_i\} \ for \ j \in K^N \end{aligned} \tag{4.24}$$

where $\hat{b} = b - \sum_{j \notin K^*} x_j(\lambda^*)$ and

$$\begin{aligned} K^Z = K^* \cap Z, \quad & b_l = \hat{b} - \textstyle\sum_{j \in K^Z} u_j \\ K^N = K^* \cap N, \quad & b_u = \hat{b} - \textstyle\sum_{j \in K^Z} l_j \end{aligned} \tag{4.25}$$

$(P3)$ is quite similar to a strictly concave problem (P2) without the objective function, and the existence of K^Z looses the knapsack constraint to be bounded. The feasibility of $(P3)$ gives a chance to find a restricted solution $\mathbf{x}^R$ in (4.23). If $\mathbf{x}^R$ is found, we are able to find a global solution distributing the value of $\hat{b} - \sum_{j \in K^N} x_j$ to x_j for $j \in K^Z$ via Algorithm 4.1. Otherwise, we find a feasible solution of $(P3)$ allowing a single x_j for $j \in K^N$ to be a value within $[l_j, u_j]$ in the mind of Theorem 4.8 and then distribute the remaining value to x_j for $j \in K^Z$ by Algorithm 4.1. All these steps are presented in Algorithm 4.2, and it has $O(n)$ time complexity if step 4 is not used.

The problem $(P3)$ can be exactly solved in pseudo polynomial time by dynamic programming or branch and bound techniques if it is feasible, but a candidate solution can be simply found by our $O(n)$ time version Algorithm 4.3. Even in the case that Algorithm 4.3 fails to find $\mathbf{x}^R$, its feasible solution roles a good starting solution for dynamic programming or

Algorithm 4.2 Try to find the best $\mathbf{x}^{U*}$ in case of $K^* \cap Z \neq \emptyset$ and $K^* \cap N \neq \emptyset$,

```
1. Find $K^Z = K^* \cap Z$ and $K^N = K^* \cap N$
2. Set $\mathbf{x}^{U*} = \mathbf{x}(\lambda^*)$ and get $b_l$ and $b_u$ in (4.25)
3. Solve $(P3)$ to get $\mathbf{x}$ using Algorithm 4.3
4. If $\mathbf{x} \notin \mathbf{x}^R$, solve $(P3)$ to get $\mathbf{x}$ using dynamic programming or branch and bound
   techniques based on $\mathbf{x}$ from step 3
5. Finish with a solution
   a) Set $\mathbf{x}^{U*} = \mathbf{x}$, $i = 1$, $g_u = g(\lambda^*) - \sum_{j \in K^N} x_j^{U*}$, and $s = \sum_{j \in K^Z} u_j$
   b) Use step 3 of Algorithm 4.1 to find $x_j^{U*}$ for $j \in K^Z$
   Note that if $\mathbf{x} \in \mathbf{x}^R$, $\mathbf{x}^{U*}$ is global optimum, otherwise $\mathbf{x}^{U*}$ is just a feasible upper
   bound solution
```

branch and bound techniques in step 4 of Algorithm 4.2.

Algorithm 4.3 is a modified version of Algorithm 2.11 for bounded knapsack constraint, and Algorithm 2.11 is an improved version of $O(n\ log\ n)$ time Algorithm 2.10 of [83] (see Section 2.4.1). It solves $(P3)$ with objective function of

$$Min\ \frac{1}{2} \sum_{j \in K^N} d_j x_j^2$$

finding a local optimum solution. All solutions are at its extreme point except for possibly one x_k as in step 4.a, and it is obtained in a similar sense of $x_j(\lambda)$ for $j \in N$ in (4.8).

As we stated earlier in Section 4.2.2, the probability that K^* includes multiple index of N may be low in practice because a special structure of (P) is required to have exactly same multiple t_j for $j \in Z \cup N$. Thus, we may have a little chance to face the global case 3 unless (P) has a specific structure.

Case 4 $K^* \subseteq N$

We have discussed this case in Section 4.2.2 mentioning that this is the *weakness* of CBS since it may not be able to find a global optimum solution in this case. However, we still

Algorithm 4.3 $O(n)$; A modified version of Algorithm 2.11 for $(P3)$

```
Define a median function: $[L, k, U] = argmed\{s_j : j \in I\}$
where $k$ is the index such that $s_k$ is the $\lceil n/2 \rceil$th $s_j$ for $j \in I$, and $s_i \le s_k \le s_j$ for $i \in L$ and for $j \in U$

  1. Set $\mathbf{s} = \{-d_j l_j \ \forall j\}$ and $U = \{1, 2, ..., n\}$
  2. Find $[L, k, U] = argmed\{s_j : j \in U\}$
  3. Get $g = \sum_{j \in L} l_j + \sum_{j \in U} u_j$
  4. Iterate until stopping criterion is satisfied
     If $b_u - g < l_k$ ($k^*$ is in $L$), set $x_U = u_U$, $k_{old} = k$, $[L, k, U] = argmed\{s_j : j \in L\}$,
         $g = g - \sum_{j \in \{k\} \cup U} l_j + \sum_{j \in \{k_{old}\} \cup U} u_j$
     Elseif $b_l - g > u_k$ ($k^*$ is in $U$), set $x_L = l_L$, $k_{old} = k$, $[L, k, U] = argmed\{s_j : j \in U\}$,
         $g = g + \sum_{j \in \{k_{old}\} \cup L} l_j - \sum_{j \in \{k\} \cup L} u_j$
     Else
       a) If $b_l - g \le l_k$, then $x_k = l_k$
          Elseif $b_u - g \ge u_k$, then $x_k = u_k$
          Else $x_k = \begin{cases} b_u - g & if \ |b_u - g| > |b_l - g| \\ b_l - g & otherwise \end{cases}$
       b) Finish algorithm with
          $x_j = \begin{cases} l_j & for \ j \in L \\ u_j & for \ j \in U \end{cases}$
```

have chances to find a global solution if $g(\lambda^*)$ or $g_l(\lambda^*)$ is *zero* or if we can find a restricted solution $\mathbf{x}^R$ in (4.23) solving (P2) due to Theorem 4.10. Our $O(n)$ time Algorithm 2.11 finds a local optimum for the strictly concave problem (P2), but it may tend not to be in $\mathbf{x}^R$ so the solution may not satisfy the necessary condition in Theorem 4.10. Thus, we may have to use a nonlinear time algorithm such as dynamic programming or branch and bound techniques to try to find a solution in $\mathbf{x}^R$. This is presented in step 1-2 of Algorithm 4.4.

Even if we fail to find a $\mathbf{x}^R$, we still have a chance to find a possibly local optimum of (P). A way to attempt to find a local optimum is presented in step 4-5 of Algorithm 4.4. The idea is based on the fact that we know λ^*. If one x_m for $m \in K^* \subseteq N$ should be in (l_j, u_j) to satisfy the knapsack constraint by a local optimum condition Theorem 4.8, the

Algorithm 4.4 Try to find the best $\mathbf{x}^{U*}$ in the case of $K^* \subseteq N$

```
1. Try to solve (P2) to get $\mathbf{x}$ using Algorithm 2.11
2. If $\mathbf{x} \notin \mathbf{x}^R$, try to solve (P2) to get $\mathbf{x}$ using dynamic programming or branch and bound
   techniques based on $\mathbf{x}$ from step 1
3. If $\mathbf{x} \in \mathbf{x}^R$, then finish algorithm with $\mathbf{x}^{U*} = \mathbf{x} \rightarrow$ Global optimum
4. Set $s_{old} = \infty$ and $s = \infty$
5. Iterate for $m = 1$ to $|K^*|$
    a) Hold $x_m = -\lambda^*/d_m$
    b) Solve $(P2.2)$
```

$$
\begin{array}{lll}
(P2.2) & Min & \sum_{j \in K^* \backslash \{m\}} (l_j - x_j)(u_j - x_j) \\
 & s.t & \sum_{j \in K^* \backslash \{m\}} x_j = b - \sum_{j \notin K^*} x_j(\lambda^*) - x_m \\
 & & x_j = \{l_j, u_i\} \ for \ j \in K^* \backslash \{m\}
\end{array}
$$

```
    c) If a solution of $(P2.2)$ is found and satisfies local optimum condition (4.22), get
       a partial local optimum objective value $s = \sum_{j \in K^*} d_j x_j^2$
    d) If $s < s_{old}$, update $x_j^{U*} = x_j$ for $j \in K^* \rightarrow$ Finish with $\mathbf{x}^{U*}$, a possibly local or
       global optimum solution
6. If $s = \infty$ (no local optimum found), finish with the best solution $\mathbf{x}$ in step 1 or step 2
   $\rightarrow \mathbf{x}^{U*}$ is a feasible upper bound solution
```

value should be $x_m = -\lambda^*/d_m$, and other x_j for $j \in K^* \backslash \{m\}$ can be found solving $(P2.2)$, which is a subset-sum problem and can be exactly solvable by well developed 0-1 knapsack problem algorithms such as `COMBO` of [79]. Thus, the best local optimum solution may be found during step 4-5. However, we do not know if the local optimum is a global optimum, and even it is proved that determining whether a solution is global optimum is *NP-hard* by Sahni [115].

Finally, if we fail to find a local optimum, we may finish the algorithm with just a feasible upper bound solution that is obtained in step 1 and step 2. However, again as we stated in Section 4.2.2 and the global case 3, it is likely that K^* does not consist of multiple index of N in general because it happens only when (P) has a special structure like a subset-sum problem. Hence, if only one index $j \in K^* \subseteq N$ exists, we can find a $\mathbf{x}^{U*}$ easily by (4.20).

4.2.4. Lower index rule to develop $O(n)$ time `CBS` method

As discussed, in the global case 1 and 2, the global optimum solution can be obtained by `CBS` in $O(n)$ time if median search method in Section (2.2.2) used for strictly convex case is applied. However, in the global case 3 and 4, executing Algorithm 4.2 and 4.4 is practically not efficient since the algorithms require to solve *NP-hard* problems. On the other hand, if the maximum *bound gap* in (4.18) is expected to be acceptable, finding a feasible $\mathbf{x}^{U*}$ by adjusting the value x_j for $j \in K^*$ and allowing at most one $x_j \in (l_j, u_j)$ for $j \in K^* \cap N$ is practically efficient, and the solution guarantees the acceptable solution quality.

In this respect, we can develop `CBS` method that has the $O(n)$ time complexity with the *lower index rule*. The *lower index rule* simply switches $x_j = u_j$ to l_j from the *lower index* in K^* until the switched value satisfies the knapsack constraint. The last switched value may be adjusted in the range of $[l_j, u_j]$ to satisfy the knapsack constraint. The similar idea is presented in step 3 of Algorithm 4.1. Moreover, in the global case 3, $O(n)$ time complexity for CBS can be also achieved if Algorithm 4.2 that omits step 4 is used for the case.

4.3. Methodology and Implementation

We have shown that the dual problem (D) can be globally solved with the lower bound solution $\mathbf{x}(\lambda^*)$, even though the primal problem (P) is *NP-hard* when $\mathbf{D}$ is nonconvex. Moreover, we also showed that $\mathbf{x}(\lambda^*)$, which guarantees the lower bound, can be used to derive a feasible solution $\mathbf{x}^{U*}$ with the proof of its existence by Theorem 4.7.

Then, one question naturally arises:

"How to find λ^*?"

We revealed two important characteristics related to λ. First, it exists within $[\lambda_{min}, \lambda_{max}]$. Second, λ^* exist at the point where $g(\lambda)$ passes its root where $g(\lambda^*) = 0$. Fortunately, if (P) is in the strictly convex case, $g(\lambda)$ is a continuous non-increasing function having a single root as Figure 4.5a. So we are able to search for λ^* applying any root finding methods or one dimensional line search methods. In contrast, if (P) is in the non-strictly convex case,

$g(\lambda)$ is not continuous at breakpoint t_j as Figure 4.5b. However, we are still able to consider it as a root finding problem. According to Press and Teukolsky [108], it is clear by

> "If the function is discontinuous, but bounded, then instead of a root there might be a step discontinuity that crosses zero. For numerical purposes, that might as well be a root, since the behavior is indistinguishable from the case of a continuous function whose zero crossing occurs in between two "adjacent" floating-point numbers in a machine's finite-precision representation"
>
> in page 445, Numerical Recipe, third edition (2007)

Therefore, we can also apply any root finding techniques that is reliable to the discontinuous lines for non-strictly convex case of (P).

4.3.1. Methods

From the Section 2.2.4, the experiment results of previous studies show that pegging method seems to outperform over other methods for strictly convex case; however, as mentioned in geometric interpretation of Section 2.2.3, it is not applicable because the objective function no longer has a unique center to project onto the knapsack constraint if (P) is not strictly convex problem. Thus, among the methods considered in Section 2.2.2, only five root finding methods (bisection, sorting, median search, secant, and Newton) and interval test method are applicable for non-strictly convex case of (P). In the following sections, we propose ideal combinations of those methods that bring efficiency and reliability.

Reliability for non-continuous $g(\lambda)$

As we have considered the global case 3 and case 4 in Section 4.2.3.3, $g(\lambda)$ may not be continuous at λ^* with nonempty K^*, and the possibility of the cases naturally increases as the proportion of N and Z increase because those variables generate more discontinuous points on $g(\lambda)$ at breakpoints t_j.

Thus, if λ^* is one of t_j where $g(\lambda)$ is discontinuous, secant method that update λ connecting two last $g(\lambda_l)$ and $g(\lambda_u)$ may have difficulty to reduce the gap between $[\lambda_l, \lambda_u]$ around

λ^*. The similar case can also happen for strictly convex case as the secant method developer Dai and Fletcher [30] observed that secant method took a long iteration when a linear piece of $g(\lambda)$ crossing its root is very short. Newton method also has the same unreliability problem because it has to use secant method as a safeguard when $\lambda^* = t_j$.

Moreover, if (P) is not in strictly convex case, an additional termination criterion $\lambda_u - \lambda_l < \varepsilon_{gap}$ is required for bisection, secant, Newton methods because the termination condition $|g(\lambda)| < \varepsilon_{fea}$ that the methods use cannot be satisfied if λ^* is one of t_j. However, as we similarly considered in the bisection Algorithm 2.1, those methods may find worse lower and upper bounds or even fail because multiple distinct breakpoints t_j can exist within the last $[\lambda_l, \lambda_u]$ whose range is less than ε_{gap}, and it is possible that one of the breakpoints is λ^*, while those methods terminate iterations with just a λ between the last $[\lambda_l, \lambda_u]$ ignoring the possible fact of $\lambda^* = t_j$ for some j.

Therefore, if we consider reliability, we have to use methods that do not use the termination criterion of $\lambda_u - \lambda_l < \varepsilon_{gap}$ and handle breakpoints at its termination criteria. The methods that satisfies two conditions for reliability are sorting, median search, and Interval test methods.

Convergence accelerator; fixing algorithm

Inspired by pegging method, we can accelerate the convergence reducing the problem size every iteration with fixing Algorithm 4.5. It is essentially same as (2.14), which is used for pegging method and made for $j \in P$, but it can be also used for $j \in N$ and Z because $x_j(\lambda)$ is non-increasing function. Fixing algorithm has been implemented in multiple literature for strictly convex case. For example, Robinson et al. [111] and Cominetti et al. [25] used fixing algorithm for their Newton method, and Kiwiel [64] used it for his median search method. All of them report that fixing algorithm 4.5 speeded up the performance.

Although Cominetti et al. [25] noticed that fixing algorithm 4.5 saves computations, they did not use it for secant methods with their argument that $g(\lambda)$ "needs to be recomputed at the end point of the bracketing interval that was computed in previous iteration"; however,

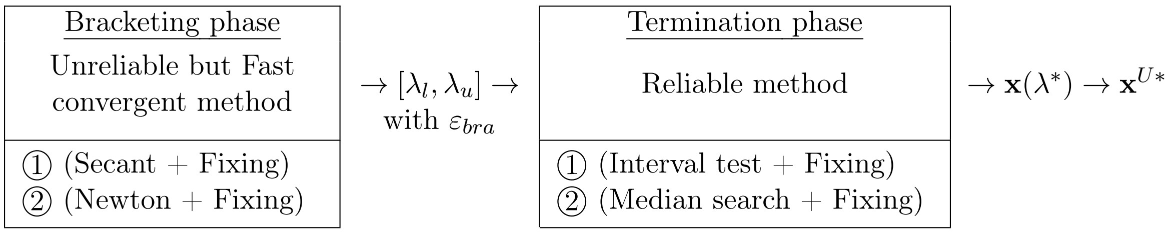
Algorithm 4.5 $O(n)$; Fixing algorithm

```
1. If $g(\lambda) > 0$, fix $x_j^* = l_j$ for $j \in \{j : x_j(\lambda) = l_j\}$
2. If $g(\lambda) < 0$, fix $x_j^* = u_j$ for $j \in \{j : x_j(\lambda) = u_j\}$
```

we do not agree with it since we can simply and have to store updated values of $g(\lambda)$ at the each bound of bracket to get secant point.

4.3.1.1. Suggested hybrid methods

Although secant and Newton methods are not reliable, through the survey of experiment results for the strictly convex case, we can observe that the methods practically converge more rapidly with less iteration than the reliable median search method (see Table 2.5). Furthermore, interval test method is reliable and spends much smaller operations in iteration part as shown in Table 2.4, but it requires a small enough bracket $[\lambda_l, \lambda_u]$ because its performance is mainly affected by the time to sort breakpoints within the bracket.


Bracketing phase
Unreliable but Fast convergent method
① (Secant + Fixing)
② (Newton + Fixing)
$\rightarrow [\lambda_l, \lambda_u] \rightarrow$ with ε_{bra}
Termination phase
Reliable method
① (Interval test + Fixing)
② (Median search + Fixing)
$\rightarrow \mathbf{x}(\lambda^*) \rightarrow \mathbf{x}^{U*}$

1. Denote "+" between methods be the hybrid of the methods.
2. Denote "Fixing" for fixing Algorithm 4.5
3. Pegging method is excluded because it is not applicable for non-strictly convex (P).
4. Bisection method is excluded because it is unreliable and has slow convergence.
5. Sorting method is excluded because it is slower than medians search method having the same iterations.

Figure 4.6.: Suggested hybrid methods for Closed-box solver (`CBS`)

Then, we can think of hybrid methods that initially converges quickly and guarantees the reliability at the last iterations as Figure 4.6. The hybrid methods consist of two phases of (a) bracketing phase and (b) termination phase, and fixing Algorithm 4.5 is equipped for all phases with the denotation of "Fixing". Thus, bracketing phase quickly gives a close enough bound of $[\lambda_l, \lambda_u]$ whose range is less than a given ε_{bra} with fast convergent method such as secant or Newton method and termination phase uses it to find λ^* exactly by reliable and efficient method such as interval test or median search method. A reliable sorting method is excluded because it is theoretically and practically requires more operations than median search method having the same iteration of median search method. Bisection is also excluded because it is neither reliable nor efficient.

Table 4.1.: Hybrid methods for strictly convex case of (P)

	Fast convergent methods	Reliable methods	Fixing
[25]	Newton		+ Fixing
[111]	Newton		+ Fixing
[137][1]	Newton + Secant		
[137][1]	Secant + Bisection		
[64, 63, 32, 101]		Median search	+ Fixing
[31]	Secant	+ Median search	

1. Wu et al. [137] used two methods sequently. e.g.) Newton for 1st iteration, Secant for 2nd iteration, Newton for 3rd iteration, and so on.
2. Ventura [133] also implemented bisection + pegging.

Various hybrid methods for strictly convex case of (P) has been implemented and suggested through a couple of papers as listed in the Table 4.1. Ironically, no one used the combinations of our suggested methods except for Dai and Fletcher [31], but they only left suggestion without implementation. Therefore, we propose new four combinations of hybrid methods in Figure 4.6 and the best combination will be determined through experiments in Section 4.3.2.

Complexity to get λ^*: $O(n)$

The complexity of our hybrid methods is determined by the choice of the methods for hybrid methods and ε_{bra}. If ε_{bra} is too small or $g(\lambda)$ has a shape that is favor to secant or Newton method, in the worst case, the complexity is $O(n^2)$ if Newton method is used, and it is not measurable if secant method is chosen.

On the other hand, if ε_{bra} is too large so the bracket $[\lambda_l, \lambda_u]$ includes all breakpoints, then, the worst case complexity is $O(n\ log\ n)$ if interval test method is selected and is $O(n)$ if median search method is utilized.

Therefore, we can keep the hybrid methods in $O(n)$ time complexity if we set $\varepsilon_{bra} = \infty$ and use median search method for termination phase. However, our experiments show that median search method is practically worse than secant method based hybrid methods.

4.3.2. Implementation

The following 7 pure algorithms and 5 hybrids methods are implemented in MATLAB.

- 7 pure methods: Pegging, Bisection, Sorting, Newton, Secant, Median search, Interval test methods

- 5 hybrid methods: Sec+Med, Sec+Sor, Sec+Int, New+Med, and New+Int

For convenient, we name the hybrid methods with the first three letters of methods with "+" meaning hybrid of two methods. Interval test method can uses $[\lambda_{min}, \lambda_{max}]$ for its initial bracket, but we used bracketing phase Algorithm 2.5 of [31], which is used for secant method. Although a hybrid method Sec+Sor is not suggested in our hybrid method in Figure 4.6, we also tested it to compare with Sec+Med because we have experienced in preliminary test that one time sorting tends to be faster than multiple time median search in practice for not too large size of array.

In the implemented code, the fixing Algorithm 4.5 is not included because it actually deteriorates the performance of codes due to MATLAB's features such as Just-in-Time compilation, copy-on-write type editing, and fast full vector computation. Appendix B.2 explains the reasons with efficient coding style in MATLAB.

The performance of median search, sorting, and interval test methods is significantly affected by the choice of sorting algorithm and median search algorithm because median search and sorting steps are the main time consuming parts in the methods. Thus, for the sorting algorithm, we used MATLAB's built-in function `sort.m`, which implemented **QUICKSORT** algorithm of Hoare [51] because it is known to be the fastest sorting algorithm in practice. For the median search algorithm, a $O(n)$ time **SELECT** algorithm of Floyd and Rivest [43] is used by compiling the `C++` standard template library `nth_element` into MATLAB, instead of MATLAB's built-in function `median.m` because the built-in function does not implemented $O(n)$ time algorithm. A detailed explanation about sorting and median search algorithms are available in Appendix B.1.

The 7 pure methods can be classified into two types:

- Root finding type: bisection, secant, and Newton methods
- Breakpoint search: sorting, median search, and interval test methods

Two types can be easily applied for non-strictly convex case of (P) to find lower and upper bound solutions and can be combined for hybrid methods.

Recall the tolerances in (2.7)

$$\begin{aligned}
\varepsilon_{gap} &> \lambda_u - \lambda_l \\
\varepsilon_{fea} &> |g(\lambda)| \\
\varepsilon_{bra} &> \lambda_u - \lambda_l \\
\varepsilon_{peg} &= \varepsilon_{fea}
\end{aligned}$$

that are used for feasibility test and iteration termination condition.

Algorithms for root finding type

As shown in Algorithm 2.1, 2.6, and 2.8, root finding type methods such as bisection, secant, and Newton methods use only ε_{fea} for termination criterion. However, an additional termination condition $\lambda_u - \lambda_l < \varepsilon_{gap}$ is required in the non-strictly convex case of (P) because $|g(\lambda^*)|$ is not always converged within ε_{fea}. So the termination criterion ε_{gap} is used first and then it is switched to ε_{fea} if no breakpoints t_j for $j \in N \cup Z$ exist between $[\lambda_l, \lambda_u]$ as in step 2.a in Algorithm 4.6. If there exists t_j with nonempty set K in step 2.a, we set λ^* to be one of the t_j assuming all t_j for $j \in K$ are identical and search for an upper bounds solution considering the global case 3 and case 4 in step 4.

Algorithm 4.6 Root finding type algorithm for indefinite case of (P)

```
1. Get necessary initial values such as breakpoints and initial $\lambda$
2. Iterate with a root finding method (bisection, secant, Newton methods)
    a) If $\lambda_u - \lambda_l < \varepsilon_{gap}$, find $K = \{j : \lambda_l \leq t_j \leq \lambda_u\}$
        i. If $K = \emptyset$, do more iteration until $|g(\lambda)| < \varepsilon_{fea}$ is satisfied in step 2.c
        ii. Else (if $K \neq \emptyset$), go to step 3
    b) Update $\lambda$ and $g(\lambda)$
    c) If $|g(\lambda)| < \varepsilon_{fea}$, go to step 5
3. Set $\lambda^* = t_j$ for a $j$ in $K$ assuming all $t_j$ for $j \in K$ are identical
4. Finish considering the global case 3 and 4
5. Finish with global optimum by the global case 2
```

Algorithms for breakpoint search type

In contrast to root finding type methods, breakpoint search type methods are reliable because it dose not assume that all t_j within $[\lambda_l, \lambda_u]$ are identical as in step 3 of Algorithm 4.6. Instead, breakpoint search type methods such as sorting, median search, and interval test methods directly use breakpoints to test its feasibility with only the termination condition of ε_{fea} and find the exact λ^* by interpolation when it finds a global optimum.

Algorithms for sorting and median search methods are described in Algorithm 4.7. The difference from the algorithms for the strictly convex case in Algorithm 2.2 for sorting method and 2.3 for median search method is at step 3-4. The binary search type methods terminate with $g(\lambda_l) > 0$ and $g(\lambda_u) < 0$, and it naturally guides the algorithm to test if $\lambda^* = \lambda_l$ at step 4 with an index set K, which is K^* in (4.17) if $\lambda^* = \lambda_l$.

Interval test method for non-strictly convex case is similar to the strictly convex case Algorithm 2.9 but has more cases to be considered as in Algorithm 4.8. The main difference is in the way to update H and G in (2.20). Since variables in $Z \cup N$ are not related with G, only H is updated when the testing breakpoint is from $Z \cup N$ as in step 3.b.ii and step 3.d.ii. The global case 3 and case 4 are also detected during step 3.b.i and step 3.d.i.

Algorithm 4.7 Breakpoint search type algorithm for indefinite case of (P)

```
1. Get necessary initial values such as breakpoints and initial $\lambda$
2. Iterate until all break points are excluded by binary search (sorting, median search
   methods)
3. Find $K = \{j : \lambda_l = t_j\}$
4. If $K \neq \emptyset$, get $g = g_u + \sum_{j \in K}(l_j - u_j)$
    a) If $g < -\varepsilon_{fea}$,
        i. Set $\lambda^* = \lambda_l$
        ii. Finish algorithm considering the global case 3 and 4
    b) Else set $g_u = g$
5. Global case 2
   Interpolate to get $\lambda^* = \lambda_l + g_u(\lambda_u - \lambda_l)/(g_u - g_l)$ and finish with $\mathbf{x}^* = \mathbf{x}(\lambda^*)$
```

Algorithms for hybrid methods

In our suggested hybrid methods in Figure 4.6, two types are combined taking advantages of initially fast convergence performance of root finding type methods and reliability of breakpoint search type methods. Thus, we can create hybrid methods pairing one of root finding type Algorithm 4.6 and one of breakpoint search type Algorithm 4.7 or interval test Algorithm 4.8. Since root finding type algorithms tend to converge slowly around λ^* as more variables in $Z \cup N$ exist, we need to replace ε_{gap} with a not too small value of ε_{bra} (we used 0.1 for experiments).

4.4. Experiments

Because we implemented 12 methods (7 pure algorithms and 5 hybrids methods), we first took extensive experiment to pick the best one before comparing with other methods such

Algorithm 4.8 Interval test algorithm for indefinite case of (P)

Given that $[\lambda_l, \lambda_u]$ $(\lambda_u - \lambda_l < \varepsilon_{bra})$, λ $(= \lambda_l$ or $\lambda_u)$, $x(\lambda)$, and $g = g(\lambda)$, and breakpoints $T = \{-d_j u_j, -d_j l_j, t_j\}$ from bracketing phase in Figure 4.6

```
1. Get $I = \{i : l_i < x(\lambda) < u_i\}$, $H = \sum_{j \notin I} x_j(\lambda) - b$, and $G = \sum_{j \in I}(1/d_j)$
2. Get $R = \{j : \lambda_l \le T_j \le \lambda_u\}$ and $T = sort(T_j : j \in R)$ in ascending order where $T$ is
   sorted $T_j$ for $j \in R$
3. If $g \ge 0$, $i = 1$, and iterate while $i \le |T|$
     a) If $G \ne 0$, set $\hat{\lambda} = H/G$
          i. If $\hat{\lambda} \le T_i$, then go to step 5
     b) If $T_i = -d_j u_j$ for $j \in P$, update $H = H - u_j$ and $G = G + 1/d_j$
        Elseif $T_i = -d_j l_j$ for $j \in P$, update $x_j^* = l_j$, $H = H + l_j$ and $G = G - 1/d_j$
        Else (if $T_i = t_j$ for $j \in N \cup Z$), find $K = \{j : T_i = t_j \ for \ j \in N \cup Z\}$
            and get $g = \sum_{j \in K}(l_j - u_j)$ and $g_u = H - T_i \cdot G + g$
          i. If $g_u < \varepsilon_{fea}$, set $\hat{\lambda} = T_i$
             A. If $K \cap Z \ne \emptyset$, go to step 7
             B. Else go to step 6
         ii. Set $i = i + |K|$, $H = H + g$ and $x_j^* = l_j$ for $j \in K$
   Else $(g < 0)$, $i = |T|$, and iterate while $i \ge 1$
     c) If $G \ne 0$, set $\hat{\lambda} = H/G$
          i. If $\hat{\lambda} \ge T_i$, then go to step 5
     d) If $T_i = -d_j u_j$ for $j \in P$, update $x_j^* = u_j$, $H = H + u_j$ and $G = G - 1/d_j$
        Elseif $T_i = -d_j l_j$ for $j \in P$, update $H = H - l_j$ and $G = G + 1/d_j$
        Else (if $T_i = t_j$ for $j \in N \cup Z$), find $K = \{j : T_i = t_j \ for \ j \in N \cup Z\}$
            and get $g = \sum_{j \in K}(u_j - l_j)$ and $g_l = H - T_i \cdot G + g$
          i. If $g_l > -\varepsilon_{fea}$, set $\hat{\lambda} = T_i$
             A. If $K \cap Z \ne \emptyset$, go to step 7
             B. Else go to step 6
         ii. Set $i = i - |K|$, $H = H + g$ and $x_j^* = u_j$ for $j \in K$
4. (at leftmost or rightmost interval) set $\lambda^* = H/G$ and go to step 5
5. Global case 2 $\rightarrow$ finish with $\lambda^* = \hat{\lambda}$ and $\mathbf{x}^* = \mathbf{x}(\lambda^*)$
6. Global case 4 $\rightarrow$ finish with $\lambda^* = \hat{\lambda}$ and try to find global optimum by Algorithm 4.2
7. Global case 3 $\rightarrow$ finish with $\lambda^* = \hat{\lambda}$ and try to find global optimum by Algorithm 4.4
```

as CPLEX. So experiments were done in the order of

$$\textit{Test } 12 \textit{ methods} \rightarrow \textit{Pick the best one} \rightarrow \textit{Compare with others}$$

To find the best method, we need good test problems. So we employed six small and three large coefficient problems, which have been used in literature, and the problems are randomly generated by the MATLAB's built-in uniform random number generator `rand.m`. Once the best method is selected, the similar settings of random problems are tested to compare it with a global [23], a local [132], and two commercial (CPLEX and MATLAB) solvers.

All experiments are conducted in two identical computers (Intel core i5, 3.47Ghz, 8GB RAM, WINDOWS 7, 64bit) in MATLAB 2013a (version 8.1), and MATLAB's built-in time measurement function `tic`/`toc` is used. See Appendix 4.3.2 for the choice of time measurement function in MATLAB for WINDOWS operations system.

Tolerances

As usual optimization solvers, we used the following tolerance values.

$$\begin{aligned}
\varepsilon_{gap} &> \lambda_u - \lambda_l &&= max\{1e-7, \varepsilon(|\lambda_l| + |\lambda_u|)/2\} \\
\varepsilon_{fea} &> |g(\lambda)| &&= max\{1e-7, \varepsilon(n + |b|)\} \\
\varepsilon_{bra} &> \lambda_u - \lambda_l &&= 0.1 \\
\varepsilon_{peg} &= \varepsilon_{fea} &&= \varepsilon_{fea}
\end{aligned}$$

where ε is the machine precision and λ_l and λ_u are initial values that each solver obtains. For example, bisection method sets $\lambda_l = \lambda_{min}$ and $\lambda_u = \lambda_{max}$, and secant method get the initial λ_l and λ_u from bracketing phase Algorithm 2.5.

The machine precision $\varepsilon = 2.2204e-16$ is used by MATLAB 2013a in test computers (WINDOWS 7, 64bit) for our experiments. ε_{gap} is used for bisection, secant, and Newton methods, and we obtain it by $\varepsilon(|\lambda_l| + |\lambda_u|)/2$ as recommended in (page 448, Numerical Recipe [108]) restricting it to be at least $1e-7$ to prevent unnecessary iterations from too small steps.

The second tolerance ε_{fea} that used for all methods are obtained in the similar manner, and the third tolerance ε_{peg} is set to be the same value because it has the same role. We chose $\varepsilon_{bra} = 0.1$ for not too small values that gives tight enough brackets for our hybrid methods based on our preliminary tests.

In addition to the tolerances, we use two more termination conditions for unreliable methods that may never satisfy $|g(\lambda^*)| < \varepsilon_{fea}$ with an empty K^* in the global optimum cases due to the numerical error for the large size problem such as $n = 5e6$. So the second tolerance gap $\varepsilon_{gap2} = 1e - 10$ is applied for secant and Newton methods to terminate iteration when

$$\lambda_u - \lambda_l < \varepsilon_{gap2} = 1e - 10$$

is satisfied because Newton method uses secant method as safeguard and secant method always reduces the gap to be at least 1/4 of the previous gap as in secant phase Algorithm 2.6. Furthermore, because we exactly know the maximum iteration of bisection method, we restrict the iteration to be less than its maximum iteration plus a constant 5. So it guarantees to terminates iteration when the total iteration is

$$Iteration > \left\lceil log_{0.5}\left(\frac{\varepsilon_{gap}}{\lambda_{max} - \lambda_{min}}\right)\right\rceil + 5 \qquad (4.26)$$

Test problems

Test problems are important to measure the performance of methods because experiments can be biased if a specific problems are tested. Thus, we use two groups of random problems that have small coefficients and large coefficients.

The small coefficient group was used by recent papers of Kiwiel (2007-2008, [63, 64, 65]) and Cominetti et al. (2012, [25]). The problems were initially used by Bretthauer et al. (1995, [18]) to test their algorithms to solve (P) with integer variables. He used three kinds of problems that have different degrees of correlations in coefficients and it was inspired by the observation of [80] that a budgeting problem which has strongly correlated coefficients is generally difficult to solve. Bretthauer et al. [18] also concluded that the strongly correlated problem is considerably difficult to solve (P) with integer variables, while Kiwiel [64] could not find significant difference in his experiment for (P) with continuous variables.

1. Small coefficients test problems are

   $(T1)$ Uncorrelated: $d_j,\ c_j,\ a_j \in [10,\ 25]$

   $(T2)$ Weakly correlated: $a_j \in [10,\ 25]$, $c_j,\ d_j \in [a_j - 5,\ a_j + 5]$

   $(T3)$ Strongly correlated: $a_j \in [10,\ 25]$, $c_j,\ d_j = a_j + 5$

   All lower and upper bounds are set to $l,\ u \in [1,\ 15]$ for all three problems.

The large coefficient group is first tested by Dai and Fletcher (2006, [31]), and Cominetti et al. (2012, [25]) used a similar coefficients. We used the three same random problems of [31] that mimic the sub-problems that arise in the real problems of multicommodity and SVM problems.

2. Large coefficients test problems are

   $(T4)$ Random: $a_j \in [-500,\ 500]$, $l_j \in [-1000,\ 0]$, $u_j \in [0,\ 1000]$, $d_j \in [1,\ 10^4]$, $c_j \in [-1000,\ 1000]$

   $(T5)$ Multicommodity: $a = 1$, $l = 0$, $u \in [0,\ 1000]$, $d_j \in [1,\ 10^4]$, $c \in [-1000,\ 1000]$

   $(T6)$ SVM subproblem: $a = \{-1,\ 1\}$, $l = 0$, $u = 1000$, $d_j = 1$, $c \in [-1000,\ 1000]$

All test problems are generated with the MATLAB's built-in uniform random number generator `rand.m`, and for indefinite problems, we randomly choose a certain percentage of d_j with MATLAB's built-in unique integer random number generator `randperm.m` and change its sign. Because $x_j(\lambda)$ for $j \in N$ performs like that for $j \in Z$, we do not tested the case of $d_j = 0$; thus, all our test problems with $(T1) - (T6)$ do not have variables of $d_j = 0$, that is, $Z = \emptyset$.

Parameters; initial λ and $\Delta\lambda$

The initial λ plays an important role for the performance of secant method because it can take less iteration as closer as the initial λ is to λ^* in bracketing phase Algorithm 2.5 of secant method, and the same reason is also applied to hybrid methods that utilize secant method.

We may get an expected value of λ^* based on the expected shape of $g(\lambda)$. Because $g(\lambda)$ is the non-increasing function, the easiest expected shape may be the linear line assuming the slopes of linear pieces of $g(\lambda)$ are not so different. Then, an expected $\bar{\lambda}$ can be obtained by

$$\bar{\lambda} = \lambda_{min} + (\lambda_{max} - \lambda_{min})\frac{b - b_{min}}{b_{max} - b_{min}}$$

based on b with $b_{min} = \sum_{l_j > -\infty} l_j$ and $b_{max} = \sum_{u_j < \infty} u_j$. However, the similar slope assumption is so naive, and it does not guarantee the quality of $\bar{\lambda}$.

Alternatively more sophisticated method is developed based on the observation that $g(\lambda)$ has a folded sigmoid shape (folded s shape). There are a couple of popular sigmoid functions such as cumulative distribution functions of statistics distributions, and we tried the algebraically simple folded logistic regression function

$$b = \frac{1}{1 + e^{\lambda}} \in (0, 1) \tag{4.27}$$

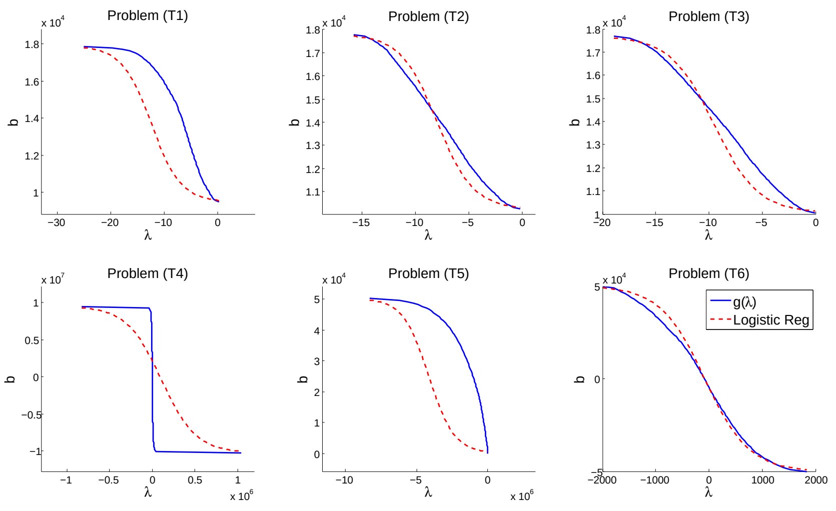
adjusting the sigmoid shape in the ranges of $[\lambda_{min}, \lambda_{max}]$ and (b_{min}, b_{max}). Then, the expected $\bar{\lambda}$ is

$$\bar{\lambda} = \left(\frac{\lambda_{max} - \lambda_{min}}{M}\right) ln\left(\frac{b_{max} - b_{min}}{b - b_{min}} - 1\right) + \frac{\lambda_{max} + \lambda_{min}}{2}$$

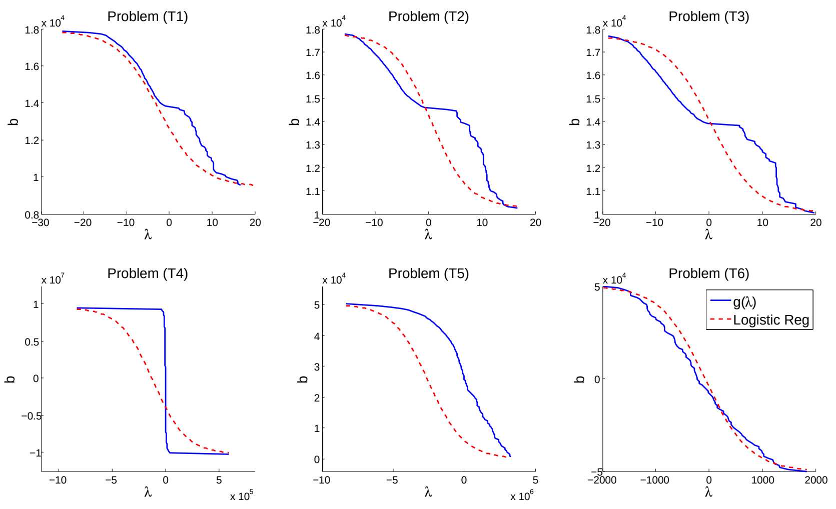
with a large enough M such that

$$M \geq -2ln\left(\frac{\pi}{1 - \pi}\right) \quad where\ \pi = \frac{\delta}{b_{max} - b_{min}}$$

Because the sigmoid function (4.27) never reaches 0 or 1, it cannot be exactly adjusted to pass the points of b_{max} at λ_{min} and b_{min} at λ_{max}. So δ is used to control gap between b_{max}



(a) Strictly positive case



(b) 50% negative d_j

Figure 4.7.: $g(\lambda)$ and folded sigmoid function

and the point of sigmoid line at λ_{min}, and the same gap is applied for the point at λ_{max}. Thus, we can control the shape of the sigmoid line with $\delta \in (0, b_{max} - b_{min})$, and the detailed derivation is available in Appendix A.6.

Examples of $g(\lambda)$ and the folded sigmoid functions are drawn in Figure 4.7 for each test problem with a given value of $\delta = (b_{max} - b_{min})/100$. The line of $g(\lambda)$ is non-increasing and has some sudden drops in indefinite problems as expected, but it is significantly different from folded sigmoid lines for some problems such as $(T4)$. The shape of $g(\lambda)$ is actually follows the cumulative distribution of breakpoints. Figure 4.8a and 4.8b show the histogram of breakpoints for size $n = 1e6$ in the strictly convex case and 50% negative d_j case respectively.

Observe the distribution of $(T6)$ in Figure 4.8. As it has the most closely bell shaped distribution, the corresponding $g(\lambda)$ in Figure 4.7 also follows most closely to the folded sigmoid function, which has a similar shape of cumulative normal distribution. We can also observe the similar relations from other problem's distributions and the corresponding lines of $g(\lambda)$. Therefore, if the distribution of breakpoints is known, an initial λ that is highly closed to λ^* can be obtained from its folded cumulative distribution function.

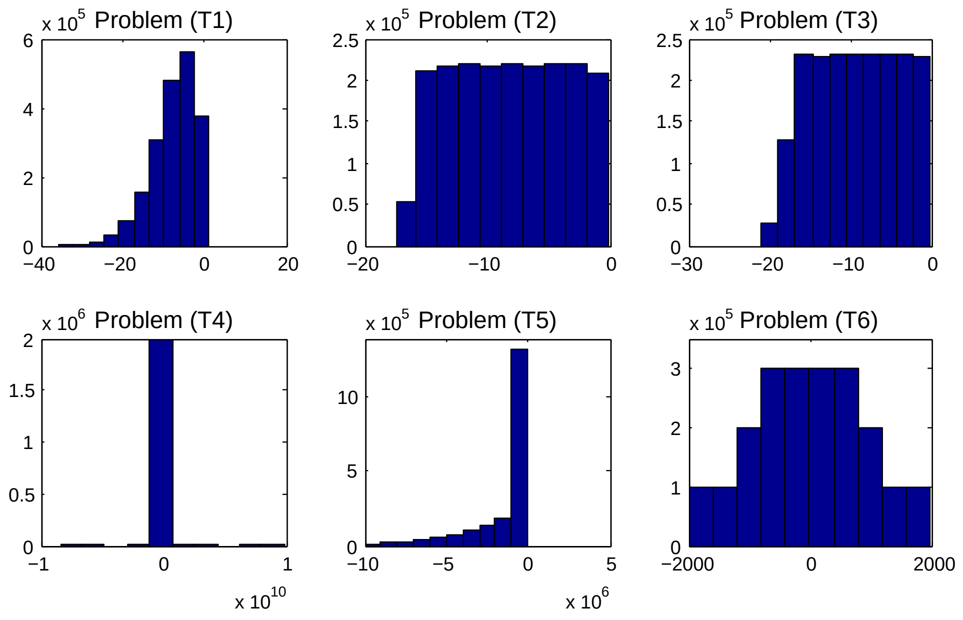
However, this idea was not applied for our implementation because the amount of computations to obtain λ_{min}, λ_{max}, b_{min}, and b_{max} is equivalent to about two iterations and, above all, finding the distribution of breakpoints is not economic.

Thus, we used the initial $\lambda = 0$ for secant method as Dai and Fletcher [31] did. They noted that it is just an arbitrary value, but it is actually numerically the best value in our transformed problem since the *zero* initial λ saves computations to get $-\lambda/d_j$ for the initial solution as

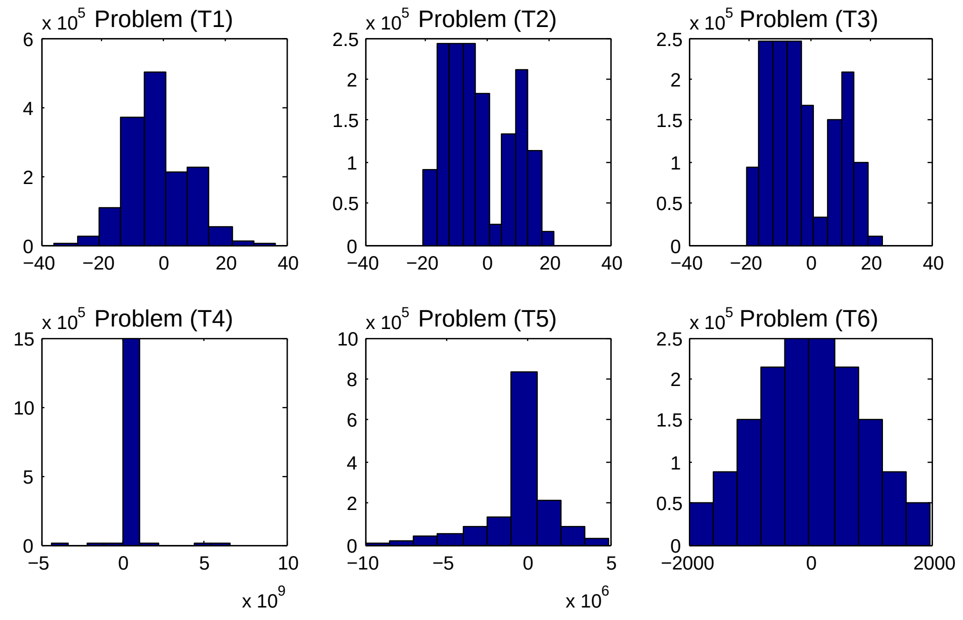
$$x_j(0) = median\{l_j, u_j, 0\} \quad j \in P$$

Moreover, secant method needs an additional parameter $\Delta\lambda$ as an initial step size. For the value, we used the same value of $\Delta\lambda = 2$ as a secant method developer [31] used.

Newton method requires an initial λ, but the *zero* initial λ does not give any benefit for convergence. Thus, as the recent study of [25], we implemented the λ in (2.12), which pegging method uses, for the initial λ of Newton and its hybrid methods. Therefore, proposed hybrid methods that uses secant and Newton methods also use the same initial values.



(a) Strictly convex case



(b) 50% negative d_j

Figure 4.8.: Histograms of breakpoints

Right hand side b

Dai and Fletcher [31] tested their problems varying the right hand side b by

$$b = b_{min} + \beta(b_{max} - b_{min}) \tag{4.28}$$

with a parameter $\beta = 0.1$, 0.3, and 0.5 and its minimum and maximum possible values b_{min} and b_{max}. That is, b is defined at a certain point of its range rather than a random number as other literatures do, and their experiment results show that the performance of secant method is significantly affected by β.

We guess it is due to the initial λ. Because λ^* is solely determined by b when other coefficients are fixed, the performance of secant method, which use the initial $\lambda = 0$, may have to take more iteration if λ^* is far from its initial λ. See the range of breakpoints in Table 4.2. Because breakpoint ranges of three small coefficient and multicommodity problems are highly biased to negative side, if secant method uses the initial $\lambda = 0$, its performance may be highly affected by the value of b.

Table 4.2.: Range of breakpoints for test problems

	Problems	Range of Breakpoints	Max. gap of breakpoints
Small coefficient	($T1$) Uncorrelated	$[-36.5, 1.5]$	38
	($T2$) Weakly correlated	$[-44.5, 2.5]$	47
	($T3$) Strongly correlated	$[-43.5, 1.5]$	45
Large coefficient	($T4$) Random	$[-\infty, \infty]$	Undetermined
	($T5$) Multicommodity	$[-100001000, 1000]$	100002000
	($T6$) SVM	$[-2000, 2000]$	4000

The suggested hybrid methods that use secant method for its bracketing phase also has a similar negative effect. Therefore, we test random problem setting the right hand side as (4.28) with $\beta = 0.2$, 0.4, 0.6, and 0.8.

Range of breakpoints

Another reason why we test small and large coefficient problems is to compare the performance on the different breakpoint ranges. As in Table 4.2, large coefficient problems have much larger range of breakpoints than small coefficient problems. Because bisection method reduces the range by half every iteration, it is quite obvious that its iteration is greater for large coefficient problems.

Moreover, we may expect that the performance of the proposed hybrid methods and interval test method would be better if breakpoints are distributed in a larger range because the larger the range, the higher the chance that a smaller number of breakpoints remained after bracketing phase with $\varepsilon_{bra} = 0.1$ is used. Thus, we expect the the proposed hybrid methods perform better for large coefficient problems.

Global instances

We test random problems varying the percentage of negative d_j to 0%, 30%, 60%, and 90% with $Z = \emptyset$. Thus, we naturally have only the global case 1.1, 2, and 4.

For 0% case, since (P) is strictly convex problem, all methods always guarantee a global optimum by the global 1.1, and the solution of the global case 2 ($K^* = \emptyset$) can also be obtained easily by $\mathbf{x}^{U*} = \mathbf{x}(\lambda^*)$.

For indefinite cases of $K^* \subseteq N$ in the global case 4, we implement "*lower index rule*" in codes instead of Algorithm 4.4. Because our random test problems do not have a special structure like subset-sum problem, all breakpoints are distinct so the probability that K^* has multiple index is very low and was *zero* in the 9000 test problems as Table 4.3. Thus, our codes actually found upper bound solution changing only $x_k^{U*} = g(\lambda^*)$ for $k \in K^*$ from $x_k(\lambda^*) = u$ as (4.20) in the global case 4.

Table 4.3.: Instances by size of K^* and % of negative d_j

$\lvert N\rvert/n$	30%	60%	90%	Total
Global	2069(69.0%)	1023(34.1%)	38(1.3%)	3130 (34.8%)
$\lvert K^*\rvert = 1$	931(31.0%)	1977(65.9%)	2962(98.7%)	5870 (65.2%)

1. Percentage in parenthesis is the proportion of instances in % of negative d_j.
2. Note that sum of proportions of global cases and $|K^*| = 1$ cases are all 100% for each % of negative d_j.

Chance to find global optimum

Table 4.3 also reveals that the chance to find global optimum naturally increases as the number of negative d_j decreases as it can be naturally expected because the non-global cases occur only when λ^* is one of breakpoints of variables in $j \in N$.

4.4.1. Selecting the best method

This section shows experiment results of 7 pure methods and 5 hybrid methods. The experiments are conducted in various environments:

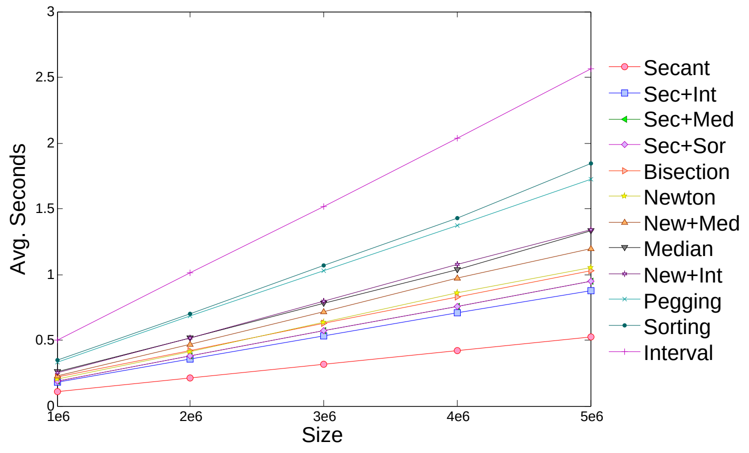
- 4 percentages of negative d_j : 0%, 30%, 60%, and 90%
- 5 sizes of n: $1e6$, $2e6$, $3e6$, $4e6$, and $5e6$
- 6 problems: $(T1)$, $(T2)$, $(T3)$, $(T4)$, $(T5)$, and $(T6)$
- 4 locations of right hand side b by $\beta = 0.2$, 0.4, 0.6, and 0.8
- 25 random problems for each instance

Thus, total $12,000$ $(= 4 \times 5 \times 6 \times 4 \times 25)$ random problems are tested. In addition to the experiment results, we also consider the reliability issue for bisection, secant, and Newton methods, and the best method is picked regarding overall performance through experiment results.

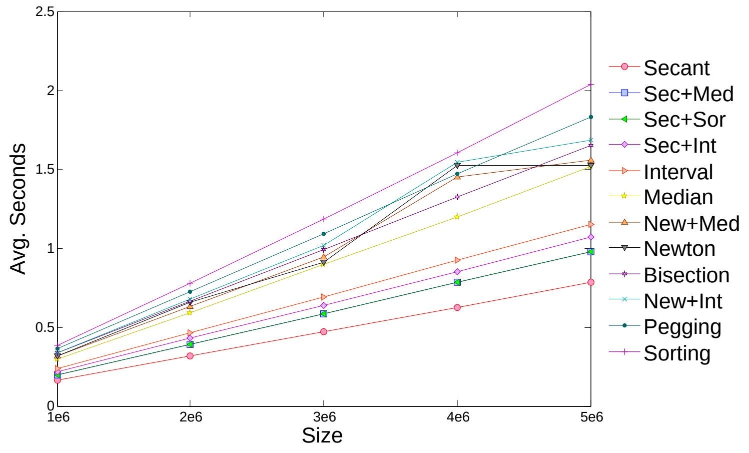
Performances in strictly convex case

The average speed of 12 methods for strictly convex case of (P) are presented in Figure 4.9 by size. Although some methods have theoretically non linear complexity, all methods have almost linear time growth by size having negligible curvature, and only Newton and its hybrid methods have a peak at $n = 4e6$ for the large size coefficient problems in Figure 4.9b.

Clearly secant method is the fastest method for all sizes and all test problems (see Figure 4.9 and 4.10). Ironically, Table 4.4 shows that secant and its hybrid methods are on top rankings having about 2 times faster speed than Newton and its hybrid methods, but their iteration is about 2 times more. It is because Newton method takes much more operations to update λ computing the slope in (2.18) than secant method. This is also observed by Cominetti et al. [25] that Newton method tends not to reflect its fewer iteration to speed.



(a) Small coefficient problems



(b) Large coefficient problems

Figure 4.9.: Average speed by size for strictly convex case

Table 4.4.: Average seconds and average iteration by size for strictly convex case

	Small coefficient problem							Large coefficient problem					
Size	1e6	2e6	3e6	4e6	5e6	Iter.	Size	1e6	2e6	3e6	4e6	5e6	Iter.
Secant	0.11	0.21	0.32	0.42	0.52	11.6	Secant	0.16	0.32	0.47	0.63	0.78	13.2
Sec+Int	0.18	0.36	0.54	0.71	0.88	9.3	Sec+Med	0.20	0.39	0.58	0.78	0.98	13.1
Sec+Med	0.19	0.38	0.57	0.75	0.95	19.1	Sec+Sor	0.20	0.39	0.58	0.78	0.98	13.1
Sec+Sor	0.19	0.38	0.57	0.75	0.95	19.1	Sec+Int	0.21	0.43	0.64	0.85	1.07	11.9
Bisection	0.22	0.42	0.63	0.83	1.02	28.2	Interval	0.23	0.46	0.69	0.92	1.15	7.2
Newton	0.20	0.41	0.64	0.86	1.05	5.6	Median	0.30	0.59	0.89	1.20	1.52	22.6
New+Med	0.23	0.47	0.72	0.97	1.20	5.4	New+Med	0.31	0.63	0.94	1.45	1.56	7.7
Median	0.26	0.52	0.78	1.04	1.33	22.6	Newton	0.32	0.66	0.91	1.52	1.52	8.8
New+Int	0.25	0.52	0.80	1.08	1.34	5.3	Bisection	0.34	0.66	0.99	1.32	1.65	45.5
Pegging	0.34	0.68	1.03	1.37	1.72	8.1	New+Int	0.34	0.68	1.02	1.54	1.68	7.4
Sorting	0.34	0.70	1.07	1.43	1.84	22.6	Pegging	0.36	0.73	1.09	1.47	1.83	7.2
Interval	0.50	1.01	1.52	2.03	2.56	4.0	Sorting	0.39	0.78	1.19	1.61	2.04	22.6

* Lists are sorted in total time for each size of coefficient.
* Iteration for interval test method is only for bracketing phase Algorithm 2.5 of [31]
* Range of speed and iterations are in Appendix Table A.3 and Table A.4.

Sorting and median search methods have the same iterations as designed but speed is discrepant due to the difference in sorting and median procedure. However, the speeds of Sec+Med and Sec+Sor are quite similar because operations for one time sorting for a small size array is competitive to that of multiple times of median search procedure.

For small coefficient problems, bisection method is on the middle ranking having faster speed than Newton and its hybrid methods, while it performs worse for large coefficient problems. It is because the ranges of dual domain for small coefficient problems are much smaller than large coefficient problems as shown in Table 4.2 so bisection method takes about 1.6 (= 45.5/28.2) times more iterations for large coefficient problems to reduce the larger range of dual domain by half. Thus, the performance of bisection method is directly affected by the size of feasible domain of dual variable.

Contrast to the experiment results of literature, the performance of pegging method is quite bad. It is because it takes more computations in each iteration as in Table 2.4 and MATLAB's fast vector computation feature. As in Table 4.4, pegging method take about half average iteration than top 5 fast methods for small coefficient problems and smallest iteration than all other methods for large coefficient problems. This behavior is consistent to results of literature; however, results of speed are conflicted. We believe it is because MATLAB has specially fast performance in full vector computations that other methods enjoy without fixing Algorithm 4.5, while pegging method consumes time to peg (memory releasing in our code) variables every iterations.

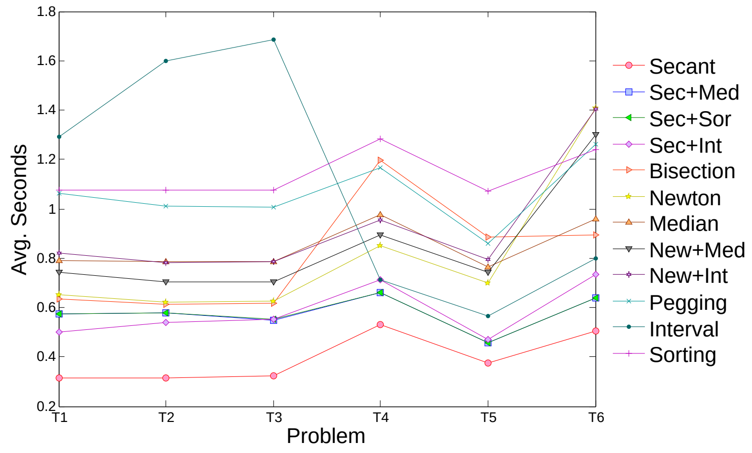


Figure 4.10.: Average speed by problem

Table 4.5.: Average seconds by problem

Test problems	T1	T2	T3	Test problems	T4	T5	T6
Secant	0.32	0.31	0.32	Secant	0.53	0.37	0.51
Sec+Int	0.50	0.54	0.55	Sec+Med	0.66	0.46	0.64
Sec+Med	0.57	0.58	0.55	Sec+Sor	0.66	0.46	0.64
Sec+Sor	0.58	0.58	0.55	Sec+Int	0.71	0.47	0.73
Bisection	0.64	0.61	0.62	Interval	0.71	0.57	0.80
Newton	0.65	0.62	0.63	Median	0.97	0.76	0.96
New+Med	0.74	0.70	0.70	New+Med	0.90	0.74	1.30
Median	0.79	0.79	0.78	Newton	0.85	0.70	1.41
New+Int	0.82	0.78	0.78	Bisection	1.20	0.89	0.89
Pegging	1.06	1.01	1.01	New+Int	0.96	0.79	1.40
Sorting	1.08	1.07	1.08	Pegging	1.17	0.86	1.26
Interval	1.29	1.60	1.69	Sorting	1.28	1.07	1.24

* Lists are sorted in total time for each size of coefficient.

* Ranges of speed are in Appendix Table A.3 and Table A.4.

Table 4.4 and 4.5 show that interval test method is the slowest method for small coefficient problems in general, while for for large coefficient problems, its ranking is high and just below the secant and its hybrid methods. It is because the number of breakpoints sorted after bracketing phase is 7.2 times less for large coefficient problems as Table in 4.6.

Table 4.6 also supports the expectation in Section 4.4 that hybrid methods have less number of breakpoints after bracketing phase for large coefficient problems, but it does not significantly affect to rankings for secant method based hybrid methods although average breakpoints are 1677.5 times less for large coefficient problems since they are on top rankings for both problem groups regardless of the number of breakpoints. Newton method based hybrid methods also cannot take the benefit because there are only few breakpoints remained after bracketing phase by Newton method for all problems.

Table 4.6.: Average number of breakpoints after bracketing phase by problem

Test problems	T1	T2	T3	T4	T5	T6	(T1+T2+T3)/ (T4+T5+T6)
Interval	2889967.4	3542640.8	3831648.7	328940.5	329518.8	767143.9	7.2
Secant based hybrid	15599.6	4913.1	9.4	11.1	0	1.1	1677.5
Newton based hybrid	0.92	1.4	0.8	0.016	0	1.3	2.5

Cominetti et al. [25] and Robinson et al. [111] noticed that newton methods rarely use safeguard, and it seems correct except for (T6) as in Table 4.7 because the total number of safeguard used over 500 instances are all less than 6; however, the proportions that safeguards are used in total iteration for SVM like problem (T6) is 57% (= 284/500) and 30% (= 161/500) on average for Newton and its hybrid methods respectively.

Table 4.7.: Total number of safeguard used for each 500 instances

Test problems	T1	T2	T3	T4	T5	T6
Newton	1	6	3	5	3	284
New+Med, New+Int	1	4	3	2	2	161

Performances in indefinite case

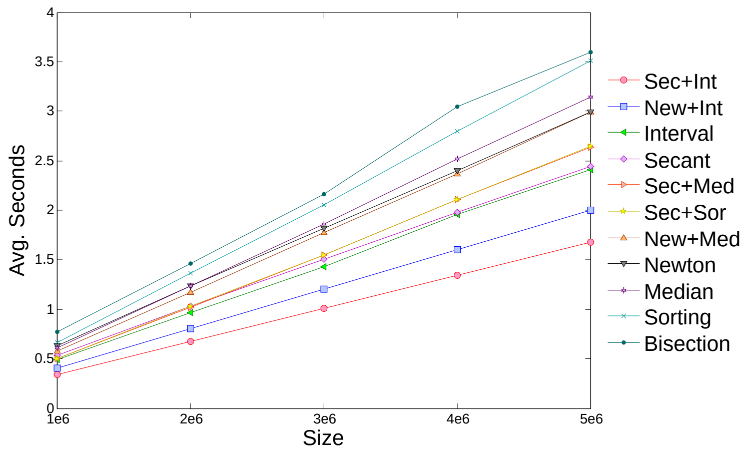
Except for pegging method, 11 methods are tested for indefinite case. Similarly to the strictly convex case, all methods have linear growth by size although sorting and Newton methods have nonlinear complexity, and only bisection methods has a little curvature at $n = 4e6$.

Hybrid methods are generally on top rankings (Figure 4.11 and Table 4.8), and the results show that interval test method has strong point for indefinite case because all three methods that employ interval test method occupy the first three rankings for small coefficient problems, and, for large coefficient problem, interval test method has the top speed and Sec+Int is followed.

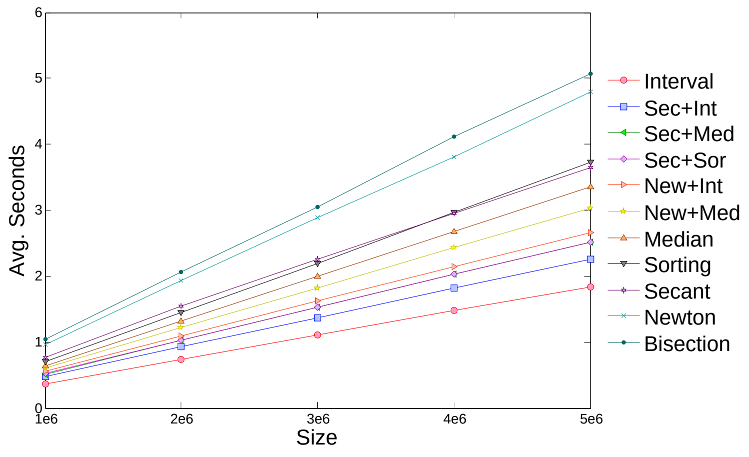
Most pure methods such as Newton, median search, sorting, and bisection methods are on the bottom ranks, but secant method is on fourth rank for small coefficient problem. Bisection method is placed on the lowest rank for both problem groups although the range of dual variable λ is narrower for small coefficient problems as in Table 4.2.

The order of rankings on size are almost identical in the respect to problems (Figure 4.12 and Table 4.10) and the percentage of negative d_j (Figure 4.13 and Table A.8). Thus, the performances are quite consistent through size, problem, and the proportion of negative d_j.

Table 4.8 shows that the rankings follow the number of iteration, but only interval test method for small coefficient problems is inconsistently on the third rank although it has the smallest iteration. It is because the feasible domain of λ is too narrow (Table 4.2) to efficiently filter out breakpoints via the bracketing phase Algorithm 2.5 of [31]. So Sec+Int is placed on the first rank for small coefficient problems filtering out more breakpoints with $\varepsilon_{bra} = 0.1$; however, it is downed to the second rank followed after interval test method for the large coefficient problems because $\varepsilon_{bra} = 0.1$ is relatively small in that case resulting in a few number (maximum 166 in Table A.7 and 6.4 on average in Table 4.9) of breakpoints. In other words, Sec+Int takes almost double (= 13.5/7) iteration of interval test method to



(a) Small coefficient problems



(b) Large coefficient problems

Figure 4.11.: Average speed by size for indefinite case

reduce the number of breakpoints, and it is less efficient than interval test method for large coefficient problems.

Table 4.9 also shows the average number of breakpoints after bracketing phase. The average number of breakpoints for interval test method in both groups of problems are very big, but it is too big to have efficiency in small coefficient problems and is small enough to have efficiency in large coefficient problems. Therefore, we can improve Sec+Int to have the best performance controlling ε_{bra} that results the optimum balance between the number of iterations in bracketing phase and the number of breakpoints to be sorted for interval test

Table 4.8.: Average seconds and average iteration by size for indefinite case

	Small coefficient problem							Large coefficient problem					
Size	1e6	2e6	3e6	4e6	5e6	Iter.	Size	1e6	2e6	3e6	4e6	5e6	Iter.
Sec+Int	0.34	0.68	1.01	1.34	1.67	10.0	Interval	0.37	0.74	1.10	1.47	1.83	7.0
New+Int	0.40	0.80	1.20	1.60	2.00	11.5	Sec+Int	0.47	0.93	1.37	1.82	2.25	13.5
Interval	0.49	0.96	1.43	1.95	2.41	3.9	Sec+Med	0.52	1.03	1.53	2.03	2.52	15.7
Secant	0.54	1.03	1.50	1.98	2.45	18.8	Sec+Sor	0.52	1.03	1.53	2.03	2.52	15.7
Sec+Med	0.50	1.02	1.55	2.10	2.64	19.3	New+Int	0.55	1.09	1.62	2.14	2.66	13.7
Sec+Sor	0.50	1.03	1.55	2.11	2.65	19.3	New+Med	0.61	1.22	1.82	2.43	3.03	17.1
New+Med	0.58	1.17	1.77	2.36	2.99	20.0	Median	0.65	1.32	1.99	2.68	3.35	22.4
Newton	0.63	1.23	1.82	2.40	2.99	21.2	Sorting	0.70	1.45	2.19	2.97	3.72	22.4
Median	0.60	1.23	1.86	2.52	3.15	22.4	Secant	0.77	1.54	2.25	2.96	3.65	28.7
Sorting	0.66	1.36	2.06	2.80	3.51	22.4	Newton	0.96	1.93	2.89	3.81	4.79	30.8
Bisection	0.77	1.46	2.17	3.05	3.60	33.0	Bisection	1.04	2.06	3.05	4.11	5.06	48.2

* Lists are sorted in total time for each size of coefficient.
* Iteration for interval test method is only for bracketing phase Algorithm 2.5 of [31]
* Ranges of speed and iterations are in Appendix Table A.5 and Table A.6.

Table 4.9.: Average number of breakpoints after bracketing phase

	Small coefficient problem			
% of negative d_j	0	30	60	90
Interval	3421419.0	1740051.3	1148770.9	2278400.3
Secant based hybrid	6840.7	1993.8	1597.1	2592.8
Newton based hybrid	1.0	1312.9	5169.7	13802.2
	Large coefficient problem			
% of negative d_j	0	30	60	90
Interval	475201.1	376307.2	297763.3	401030.5
Secant based hybrid	4.1	3.6	5.6	6.4
Newton based hybrid	0.4	44.5	16.2	23.1

* Ranges are in Appendix Table A.7.

in termination phase.

Interval test based methods have speed priority although the methods include $O(n\ log\ n)$ complexity sorting procedure. We believe that it is because the one time sorting step is practically not so costly up to some size of array. This is supported by performances of methods that use sorting. Sorting method that sorts entire breakpoints is ranked on the bottom for all problem groups, but Sec+Sor that sorts a part of breakpoint is on the middle upper ranking. Moreover, both methods are just followed after median search method and Sec+Med respectively meaning cost for one time sorting of not too many breakpoints is similar to the multiple times of median search.

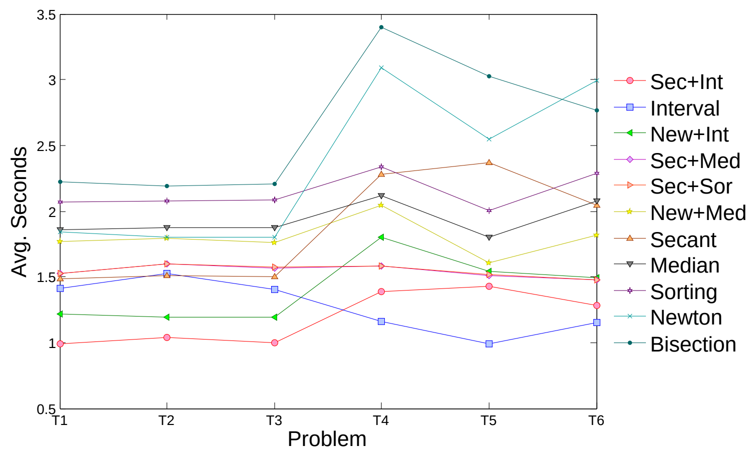


Figure 4.12.: Average speed by problem

Table 4.10.: Average seconds by problem

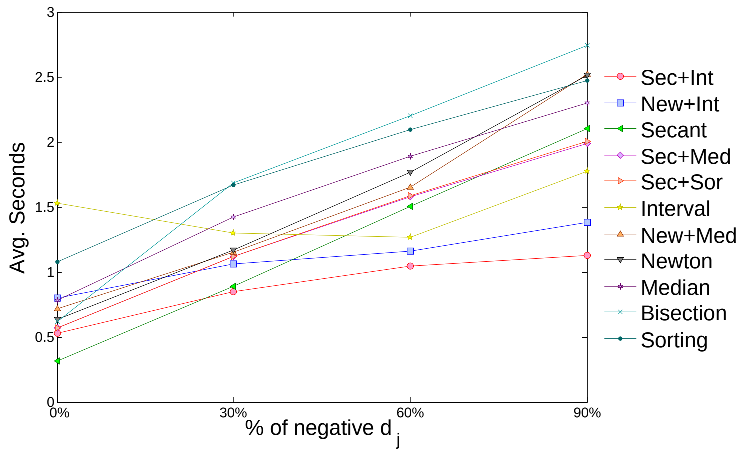
Test problems	T1	T2	T3	Test problems	T4	T5	T6
Sec+Int	0.99	1.04	1.00	Interval	1.16	0.99	1.16
New+Int	1.21	1.20	1.19	Sec+Int	1.39	1.43	1.28
Interval	1.41	1.53	1.40	Sec+Med	1.58	1.51	1.48
Secant	1.48	1.51	1.50	Sec+Sor	1.58	1.51	1.48
Sec+Med	1.52	1.59	1.57	New+Int	1.80	1.54	1.49
Sec+Sor	1.53	1.60	1.57	New+Med	2.04	1.61	1.81
New+Med	1.77	1.80	1.76	Median	2.12	1.80	2.08
Newton	1.85	1.80	1.80	Sorting	2.34	2.00	2.29
Median	1.86	1.88	1.88	Secant	2.28	2.37	2.05
Sorting	2.07	2.08	2.08	Newton	3.09	2.55	2.99
Bisection	2.22	2.19	2.21	Bisection	3.40	3.03	2.77

* Lists are sorted in total time for each size of coefficient.

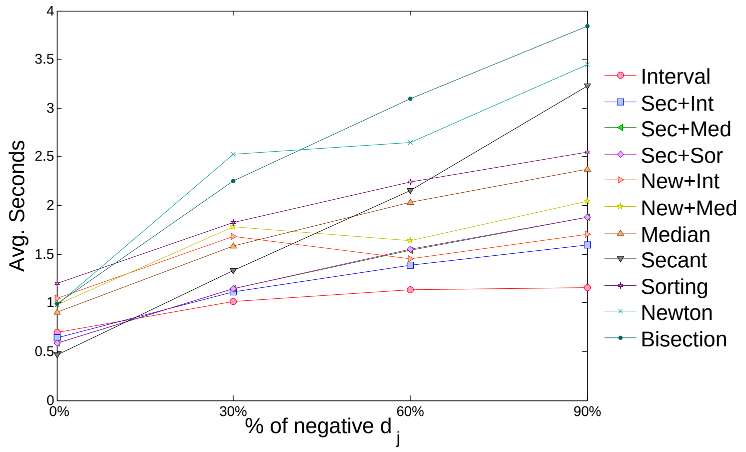
* Ranges of speed are in Appendix Table A.5.

Figure 4.12 shows that all methods performs quite evenly for small coefficient problems, but there are inconsistency for large coefficient problems. Especially methods have different performances for problem ($T5$), but hybrid methods that employ secant method have relatively stable speed.

We may think break point search type methods such as sorting, median search, and interval test methods can have better performance as the percentage of negative d_j increases because those variables generate only one breakpoint, while variables in $j \in P$ generate two breakpoints. However, it does not significantly affect the speed for sorting and median search methods as Figure 4.13 because such binary search methods can reduce the number of breakpoints by half in an iteration. For example, the number of breakpoints for no negative d_j is $2n$ and for 100% negative d_j is n. Then, sorting and median search method have just one less iteration in (P) of 100% negative d_j than in that of no negative d_j. Thus, the average number of iteration decreases slightly from 22.6, 22.6, 22.4, to 22.2 for 0 to 90% negative d_j respectively. As Figure 4.13, most methods tend to be slower as the percentage



(a) Small coefficient problems



(b) Large coefficient problems

Figure 4.13.: Average speed by % of negative d_j

of negative d_j increases, but Table 4.11, which shows the growth rate on percentage of negative d_j, shows that interval test method has the slightest growth rate in large coefficient

	Small coefficient problem				Large coefficient problem		
% of negative d_j	0-30	30-60	60-90	% of negative d_j	0-30	30-60	60-90
Sec+Int	1.06	0.66	0.27	Interval	1.09	0.38	0.09
New+Int	0.88	0.34	0.73	Sec+Int	1.59	0.91	0.68
Secant	1.91	2.05	1.99	Sec+Med	1.88	1.31	1.11
Sec+Med	1.84	1.53	1.38	Sec+Sor	1.88	1.32	1.12
Sec+Sor	1.84	1.54	1.40	New+Int	2.11	-0.77	0.83
Interval	-0.76	-0.10	1.69	New+Med	2.68	-0.48	1.34
New+Med	1.46	1.65	2.89	Median	2.28	1.50	1.14
Newton	1.78	1.98	2.49	Secant	2.85	2.76	3.56
Median	2.12	1.56	1.38	Sorting	2.10	1.39	1.01
Bisection	3.53	1.73	1.79	Newton	5.14	0.40	2.68
Sorting	1.97	1.43	1.26	Bisection	4.20	2.81	2.50

* Lists are sorted in total time for each size of coefficient.
* Growth rates that are less than 1 are colored in red indicating less than $O(n)$ time growth.
* Average seconds are available in Appendix Table A.8.

Table 4.11.: Average time growth rate by % of negative d_j

problems, and Sec+Int has less than linear growth rate for all problems and percentages. The Table also shows that bottom ranking methods and root finding type methods such as secant and Newton method tend to have high growth rate. Newton and its hybrid methods

Table 4.12.: Average percentage in use of safeguard

	Small coefficient problem				Large coefficient problem			
% of negative d_j	0	30	60	90	0	30	60	90
Newton	0.120	49.7	75.1	88.5	2.2	35.9	73.0	89.3
New+Med	0.099	18.7	31.8	41.3	1.4	10.1	42.4	66.2
New+Int	0.101	27.3	55.0	77.9	1.5	12.8	54.9	79.8

tend to use safeguard more times as the percentage of negative d_j increases as shown in Table 4.12. In the case of 90% negative d_j, Newton method uses secant method 89% of its iteration on average as a safeguard. Thus, Newton and its hybrid methods naturally take more time to compute the slope (2.18) of $g(\lambda)$, and performance of the methods always worse than secant and secant based hybrid methods.

Reliability

As we mentioned in Section 4.3.1, unreliable methods such as bisection, secant and Newton methods can have worse objective values than other reliable methods because

(a) Unreliable methods pick a breakpoint among index in K^* as step 3 of Algorithm 4.6

(b) Sometimes never satisfies feasibility with ε_{fea} due to numerical error as additional termination criteria ε_{gap2} and maximum iteration in (4.26) are required.

However, the former problem (a) does not happen in our experiments because $|K^*| \leq 1$ for all instances as in Table 4.3.

We counted the number of instances if the unreliable methods have worse lower and upper bound objective values than reliable method, that is, if the difference of objective values is greater than 0.01. Table 4.13 shows the counted number and its percentage in 9000 indefinite case instances. Fortunately, the chances of happening are not big, but it can happen relatively often for bisection method in 9%.

Table 4.13.: Total count of worse bound

Method	Lower bound	Upper bound
Bisection	807 (9%)	807 (9%)
Newton	516 (6%)	516 (6%)
Secant	83 (0.9%)	83 (0.9%)

Conclusion

For strictly convex case, secant method is obviously the fastest method, and for indefinite case, Sec+Int and interval test method are fastest for small and large coefficient problem as in Table 4.14. We can also observe from the Table that all the top ranking methods are based on secant and interval test method, and the methods are actually thought of a kind of Sec+Int because it can be secant method if we give $\varepsilon_{bra} = -\infty$ and behave like interval test method if $\varepsilon_{bra} = \infty$.

Table 4.14.: Top rankings

	Coefficients	First rank	Second rank
Strictly convex case	Small	Secant	Sec+Int
	Large	Secant	Sec+Int (fourth)
Indefinite case	Small	Sec+Int	Interval (third)
	Large	Interval	Sec+Int

For indefinite problems, Sec+Int is downed from the first rank in small coefficient problem to the second rank in large coefficient problem. Because the range $[l_j, u_j]$ for variables in large coefficient indefinite problem are large (at most 2000), $g(\lambda)$ can have relatively big jump at t_j within small range of $[\lambda_l, \lambda_u]$, and that is the case when secant method has difficulty to reduce the dual domain efficiently. Thus, last many iterations in bracketing phase of Sec+Int are consumed just to satisfy $\lambda_u - \lambda_l < \varepsilon_{bra} = 0.1$ with small dual domain reduction rate although there are already few number of breakpoints between $[\lambda_l, \lambda_u]$ are remained. So Sec+Int consumed almost double (=13.5/7, Table 4.8) iterations than interval test method remaining less than 7 (unnecessary too small) breakpoints (tables 4.9). Hence, if ε_{bra} is properly assigned to avoid the inefficient last iterations, Sec+Int can also take the first rank for large coefficient indefinite problem.

Moreover, as we have seen the almost identical performances of Sec+Sor and Sec+Med, sorting procedure is not practically worse than linear time median procedure. That means Sec+Int may not be significantly deteriorated by $O(n\ log\ n)$ sorting procedure if secant method filters out proper number of breakpoints. In addition, interval test method can find exact solution without reliability issue.

The performance of Sec+Int may be affected by the initial λ; however it actually increases the value of Sec+Int as a sub-problem solver because [25, 31, 44] experimentally show that the initial λ plays a significant role when the nonseparable QP is solved by SPG method using secant or Newton method as a sub-problem solver.

Therefore, we conclude Sec+Int is the best method among 12 tested methods and give name, Closed-box solver (`CBS`), to the method. We may let Sec+Int dynamically controls ε_{bra} looking at the remaining number of breakpoints during secant method in bracketing phase narrows $[\lambda_l, \lambda_u]$ in bracketing phase to get the optimum balance between the iteration of secant method and the number of breakpoints for interval test method. We leave it for the future research.

4.4.2. Comparison with other methods

In this section, we compare `CBS` (Sec+Int) with other selected four solvers: a global solver of Chen and Burer (2012, [23]) , a local solver of Vavasis (1992, [132]), two commercial solvers of CPLEX 12.5 and MATLAB 2013a. For detailed descriptions for compared solvers, see Appendix A.5.

Performance profiler

Dolan and Moré (2002, [33]) developed *performance profile* to compare optimization solvers, and it has been widely[1] used these days because it overcomes drawbacks of statistics such as average, ranking, and ratio in a compact graph. However, since the *performance profile* is mainly for time comparison, Dai and Fletcher [31] modified it to compare local optimum objective values. So we use *performance profile* of the version of Dai and Fletcher [31] to compare the quality of solutions of `CBS` with that of other solvers.

Suppose we have the set of solvers $\mathcal{S}$ on a test set $\mathcal{P}$ with n_p number of problems. We denote $f_{p,s}$ the objective value of a problem $p \in \mathcal{P}$ by a solver $s \in \mathcal{S}$, and let $\underline{f}_p$ and $\bar{f}_p$ be the minimal and maximal values of $\{f_{p,s} : s \in S\}$. We get the quality of $f_{p,s}$ normalizing it into a ratio as

$$Quality\ ratio: \quad r_{p,s} = \begin{cases} 0 & if\ \bar{f}_p - \underline{f}_p \leq \varepsilon \\ (f_{p,s} - \underline{f}_p)/(\bar{f}_p - \underline{f}_p) & otherwise \end{cases}$$

where $\varepsilon > 0$ is a value to admit numerical error in objective values, and we used $\varepsilon = 10^{\lfloor log_{10}(n) \rfloor} \times 10^{-6}$. Then, we draw a graph with a given factor $\tau \in [0, 0.9]$ versus probability

$$Probability: \quad \rho_s(\tau) = \frac{1}{n_p} size\{p \in \mathcal{P} : r_{p,s} \leq \tau\}$$

that the quality ratio $r_{p,s}$ is equal to or less than a factor τ for solver s. In other words, it means that a solver s can yield objective value less than $\tau \times \underline{f}_p$ in a probability of $\rho_s(\tau)$. So $\rho_s(0)$ is the probability that a solver s finds the best solution ($f_{p,s} = \underline{f}_p$ for $p \in \mathcal{P}$).

Thus, the probability ρ_s is a cumulative distribution function with a non-decreasing, piecewise constant and continuous line on τ, and a line under 1 means that the solver is not better for all problems than others.

[1][33] has been cited by 1051 papers as of July 15, 2013.

Small size problems

The five methods (`CBS`, Chen and Burer [23], Vavasis [132], `cplexqp.m` of CPLEX, and `quadprog.m` of MATLAB) are used to compare the speed and the quality of optimum values. Small sizes are tested because the global solver of Chen and Burer [23] and `quadprog.m` of MATLAB consume exponentially long time for large size problems, say over $n = 300$. The random problems are generated by

- 3 percentages of negative d_j : 20%, 50%, and 80%
- 3 sizes of n: 20, 40, and 60
- 3 small coefficient problems: $(T1)$, $(T2)$, and $(T3)$
- 3 locations of right hand side b by $\beta = 0.2$, 0.5, and 0.8
- 10 random problems for each instance

Thus, the total 810 $(= 3 \times 3 \times 3 \times 3 \times 10)$ problems are tested. Note that large coefficient problem $(T4)$, $(T5)$, and $(T6)$ could not be used because the solver of Chen and Burer [23] tends to fail to solve frequently for the problems.

The average and range of speed (in seconds) of five methods are presented in Table 4.15. `CBS` spent a fraction of time with almost no time growth, while other methods shows nonlinear time growth. Chen and Burer shows the fastest time growth, but the growth rate is actually much higher because the method reached the time limit up to 42.6% as shown in Table 4.16.

Table 4.15.: Average seconds and range for small size indefinite problems

Size	`CBS`	Vavasis	CPLEX	MATLAB	Chen&Burer
20	0.00054	0.0018	0.017	0.018	23.3
40	0.00048	0.0036	0.019	0.052	318.4
60	0.00055	0.0066	0.021	0.109	489.6
20	[0.0002, 0.008]	[0.0009, 0.013]	[0.015, 0.048]	[0.01, 0.037]	[1.013, 322.93]
40	[0.0002, 0.002]	[0.0013, 0.019]	[0.015, 0.055]	[0.034, 0.081]	[3.404, 900]
60	[0.0002, 0.002]	[0.002, 0.038]	[0.016, 0.055]	[0.074, 0.142]	[8.527, 900]

Rather than speed, we may compare the probabilities that solvers find global optimum. Since Chen and Burer theoretically guarantees the global optimum solution, we can compare global optimality comparing with the objective value of Chen and Burer. However, because Chen and Burer sometimes finds worse objective value than other solvers and overs the time limit of 15 minutes, we obtain the probabilities counting the instances that a solver's objective value is not different from the best value of all solvers. That is, the probability is

$$P[f_{p,s} - \underline{f}_p < n \times 10^{-6} : p \in \mathcal{P},\ s \in \mathcal{S}] \tag{4.29}$$

with numerical error admissible range of $n \times 10^{-6}$. The percentage values are in Table 4.16. Most solvers can similarly get global optimum in about 60%, and Chen and Burer fails to find global optimum in 2.7%, which is much less than time limit over rate 22%. But the time over rate significantly increases as size grows.

Although the upper bound solutions of `CBS` are just feasible solutions when $K^* \neq \emptyset$, the solutions are actually the best or global in 12% as shown in the last column of Table 4.16. But the rate decreases as the size grows.

Table 4.16.: Percentage of the best solution found for small size indefinite problems

Size	`CBS`	Vavasis	Cplex	Matlab	Chen&Burer	Time > 15 min. of Chen&Burer	$K^* \neq \emptyset$ and `CBS` is the best
20	67.0	66.3	72.2	62.2	95.6	0.0	17.8
40	62.6	58.1	65.9	56.7	99.3	23.3	10.7
60	58.9	55.9	61.9	54.4	97.0	42.6	8.1
Overall	62.8	60.1	66.7	57.8	97.3	22.0	12.2

Since performance profile gets cumulated distribution pooling all objective values, we cannot use the numerical error admissible range of $n \times 10^{-6}$ that is used for Table 4.16. So we use

$$\varepsilon = 10^{\lfloor log_{10}(n) \rfloor} \times 10^{-6} = 10^{-5}$$

for performance profile in Figure 4.14. Due to the different numerical error admissible range, $\rho(1) = 0.6$ for Cplex is significantly different from 66.7% in the last row of Table 4.16.

Chen and Burer, of course, is placed on the top line almost along at 1 as a global solver, while MATLAB's `quadprog.m` always yields the worst quality solutions. `CBS` is almost 10% above CPLEX for $\tau > 0.15$. That means the quality of upper bound solution of `CBS` is quite better than that of CPLEX. It may be because the small coefficients of test problems ($T1$), ($T2$), and ($T3$) yield relatively small bound gaps (4.18) for `CBS`. Test with large coefficients are followed in the next section.

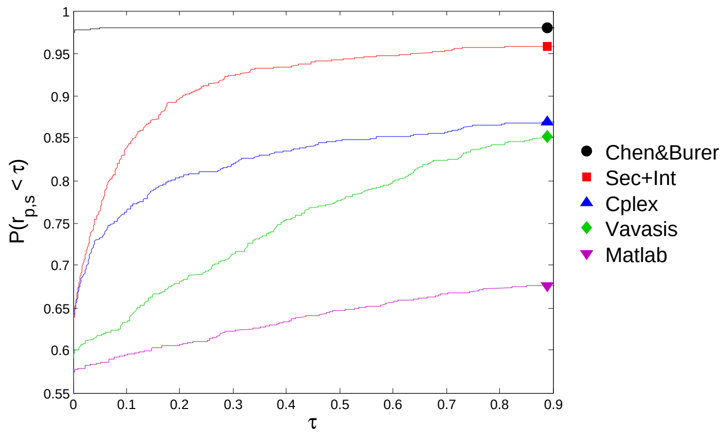


Figure 4.14.: Performance profile for small size indefinite problems

Large size problems

In addition to the small size problems, experiments with large size are also conducted without MATLAB's `quadprog.m` and the global solver of Chen and Burer since two methods consume too much time and fail to solve in many times for large size problems. The random problems are generated by

- 3 percentages of negative d_j : 20%, 50%, and 80%
- 4 sizes of n: 1000, 20000, 3000, and 4000
- 6 problems: ($T1$), ($T2$), ($T3$), ($T4$), ($T5$), and ($T6$)
- 3 locations of right hand side b by $\beta = 0.2$, 0.5, and 0.8
- 10 random problems for each instance

Thus, the total $2,160$ $(= 3 \times 4 \times 6 \times 3 \times 10)$ problems are tested. Note that we include all six test problems for this case because Chen and Burer is not tested in this large size problems.

The average speeds and ranges of solvers in seconds are presented in Table 4.17. All solvers tend to take more time in large coefficient problems, and both Vavasis and CPLEX spent approximately double time for large coefficient problems than for small coefficient problems although the most minimum times are similar for both problem groups.

Time growth rate can be computed by

$$\frac{t_{n,s}/t_{1000,s}}{(n/1000)^{Growth\ rate}} \backsimeq 1 \tag{4.30}$$

with $t_{n,s}$, the average time for solver s in the size of n, and it is presented on the middle row of Table 4.15. Growth rates are approximately identical for both sizes of coefficients in the three solvers, and `CBS` shows less than linear time growth rate (0.5). CPLEX has almost linear growth rate (1.2), but Vavasis has more than quadratic growth rate (2.3) over size.

Table 4.17.: Average seconds and range for large size indefinite problems

	Small coefficient			Large coefficient		
Size	CBS	Vavasis	CPLEX	CBS	Vavasis	CPLEX
1000	0.0006	0.9	0.12	0.0008	1.9	0.22
2000	0.0009	4.3	0.26	0.0011	9.1	0.50
3000	0.0012	11.4	0.44	0.0016	24.2	0.90
4000	0.0014	23.6	0.66	0.0019	49.2	1.30
Growth rate	0.5	2.3	1.2	0.5	2.3	1.2
1000	[0.0003, 0.003]	[0.04, 3.9]	[0.04, 0.4]	[0.0004, 0.004]	[0.05, 4.7]	[0.04, 0.8]
2000	[0.0005, 0.003]	[0.13, 22.3]	[0.09, 0.7]	[0.0005, 0.004]	[0.13, 23.1]	[0.07, 1.7]
3000	[0.0006, 0.004]	[0.27, 53.3]	[0.14, 1.3]	[0.0007, 0.006]	[0.27, 62.5]	[0.12, 4.2]
4000	[0.0008, 0.005]	[0.44, 116.3]	[0.21, 1.8]	[0.0008, 0.008]	[0.44, 129.1]	[0.17, 5.7]

Consider the probability that a solver gets the best optimum solution as (4.29). Table 4.18 shows the probabilities, and `CBS` has obviously great rate for best optimum solution. We expect that `CBS` would yield worse upper bound objective value for large coefficient problems because large coefficients naturally give larger bound gap (4.18); however, it might affect to Vavasis and CPLEX more seriously since the probabilities of Vavasis and CPLEX for large coefficient problems are significantly lower than that for small coefficient problems. Another reason may be found in the last columns for each size coefficient groups in Table 4.18. Because the probabilities that CBS finds the best when $K^* \neq \emptyset$ is greater in large coefficient problems than in small coefficient problems, it may leads the lower probabilities for other methods.

Table 4.18.: Percentage of the best solution found for large size indefinite problems

	Small coefficient				Large coefficient			
Size	CBS	Vavasis	CPLEX	$K^* \neq \emptyset$ and Best[1]	CBS	Vavasis	CPLEX	$K^* \neq \emptyset$ and Best[1]
1000	83.3	50.7	63.7	33.0	94.1	15.9	5.9	44.8
2000	88.5	49.6	54.8	39.3	94.1	16.3	5.2	41.1
3000	93.3	50.0	50.0	43.7	94.4	15.2	5.2	47.0
4000	91.5	50.0	54.1	42.2	94.4	16.3	5.6	47.4
Overall	89.2	50.1	55.6	39.5	94.3	15.9	5.5	45.1

1. $K^* \neq \emptyset$ and Best means CBS yields the best solution when $K^* \neq \emptyset$.

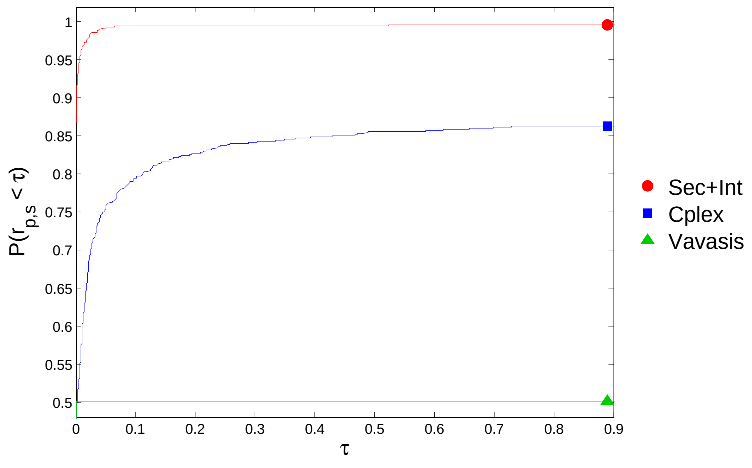
Performance profiles in Figure 4.15 also clearly show the worse performance of Vavasis and CPLEX. The two lines of the methods have larger gap from the line of CBS in large coefficient problems

For the same reason in the small size experiments, Table 4.18 is obtained with the numerical error admissible range of $n \times 10^{-6}$ and performance profiles in Figure 4.15 is obtained with

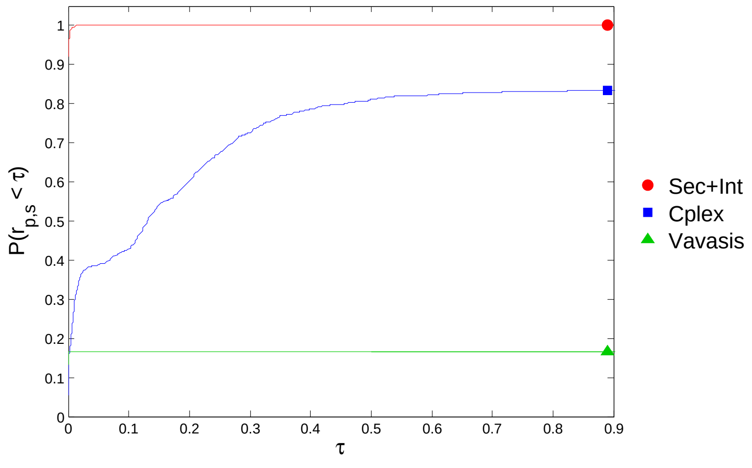
$$\varepsilon = 10^{\lfloor log_{10}(n) \rfloor} \times 10^{-6} = 10^{-3}$$

by pooling all objective values. Thus, $\rho(1) = 0.5$ for CPLEX in Figure 4.15a of small coefficient problems is discrepant from 55.6% on the last row of Table 4.18.

Performance profiles show that CBS always above Vavasis and CPLEX touching the probability of 1. Vavasis never increases its probability through all $\tau = [0,\ 0.9]$, and it means its quality of solutions is quite worse than other solvers.



(a) Small coefficient problems



(b) Large coefficient problems

Figure 4.15.: Performance profile for large size indefinite problems

4.4.3. Experiments with GN2

Since CPLEX 12.6 was recently released on December 2013 with a new global quadratic solver, CBS is also tested with GN2 of Section 3.4.3. Table 4.19 is made based on Table 3.7 adding the results of CBS. The upper bound of CBS and CPX-G are very close with numbers almost 0%; however, somehow, all average relative errors are negative meaning that CBS finds better global optimum solution than CPX-G. In the experiment results, the global objective value of CPX-G is always greater than or equal to the objective value of CBS for all tested problem. This phenomenon might mean that CPX-G does not guarantee an exact global

Table 4.19.: Performance of CBS for GN2$(n, 10, 0, 0.5)$

	Problem size (n)	500	1,000	2,500	3,500	5,000
UB of OBG	rel. error[1]	5.62%	6.07%	5.19%	5.36%	5.05%
	cpu sec.	0.184	0.480	2.335	4.545	8.695
VAV92	rel. error[2]	1.22%	1.21%	1.25%	1.19%	1.19%
	cpu sec.	0.646	2.834	23.278	52.499	129.266
UB of CBS	rel. error[4]	-0.001%	-0.002%	-0.001%	-0.003%	-0.002%
	cpu sec.	0.0005	0.0006	0.0010	0.0016	0.0021
CPX-G	cpu sec.	0.187	0.370	0.723	2.005	3.187

	Problem size (n)	10,000	20,000	30,000	40,000	50,000
UB of OBG	rel. error	5.51%	4.84%	5.27%	5.12%	5.09%
	cpu sec.	34.1	160.7	338.5	583.5	890.5
VAV92	rel. error	1.24%	1.21%	*	*	*
	cpu sec.	815.5	5656.2	*[3]	*	*
UB of CBS	rel. error[2]	-0.0008%	-0.0006%	-0.0002%	-0.0002%	-0.0004%
	cpu sec.	0.0034	0.0063	0.0091	0.0115	0.0135
CPX-G	cpu sec.	9.891	41.421	100.451	197.155	334.716

1. Relative error is defined by $[UB(\texttt{OBG})$ - obj(CPX-G)]/abs[obj(CPX-G)]
2. Relative error is defined by [obj(VAV92) - obj(CPX-G)]/abs[obj(CPX-G)]
3. 2 hour cpu time limit or memory exceeded for all problems
4. Relative error is defined by $[UB(\texttt{CBS})$ - obj(CPX-G)]/abs[obj(CPX-G)]

solution. In the respect of speed, CBS is excessively faster than all other methods. Further performance investigation of CBS in a broader range of test problems are left for the future research.

4.5. Conclusion

A linear time algorithm to find lower and upper bounds of indefinite knapsack separable quadratic programs is presented. The relaxation of the knapsack constraint with a single Lagrange multiplier makes the dual problem completely separable, and it gives a chance to find a lower bound very efficiently in the root-finding scheme. In the experiments, the proposed hybrid root-finding algorithm performs significantly better than existing methods, and moreover, it turns out that the best methods, Sec+Int is fairly impressive in the respect of speed and solution quality through all tested problems relative to the compared solvers.

4.5.1. Future research in `CBS`

Two chances to improve `CBS`

Experiments show that `CBS` is superior to other tested solvers, but there are still two chances to improve it. One is the parameter ε_{bra} that determines the iteration to switch from secant method to interval test method. Because sorting the breakpoints for interval test method consumes more significant time as there exist more breakpoints within the bracket range, which is less than ε_{bra}, a method to determine the optimum ε_{bra} that can balance the iterations of secant and interval test methods is required. The other chance is for the quality of the upper bound solution. As discussed at the end of section 4.2.2, when multiple breakpoints from N are exactly same as λ^*, `CBS` may not determine the best upper bound solution since it is found by our suggested heuristic lower index rule. Thus, further research is required for the two chances.

Minimizing the *bound gap*

In the case that all breakpoints t_j for $j \in N$ are unique,the *bound gap* can be further minimized with a better upper bound solution with the following idea. Suppose all breakpoints t_j for $j \in N$ are unique in non-strictly convex (P), and there exists

$$k = \{j \in N : x_j^{U*} \in (l_j, u_j)\}.$$

In the case, $\mathbf{x}^{U*}$ does not guarantee local or global optimality because conditions that verify the global optimality of indefinite (P) are not known. Then, we may think of three cases for global optimum solution $x_k^* = l_k$, u_k, or x_k^{U*}. Since the third case is $\mathbf{x}^{U*}$, the other two cases can be tested with `CBS` fixing $x_k = l_k$ and u_k. Each case generates the corresponding pair of $(\lambda_L^*,\ \mathbf{x}_L^{U*})$ and $(\lambda_U^*,\ \mathbf{x}_U^{U*})$. These two paris can be obtained in relatively little computation because of two reasons: (see Figure 4.16 to help understanding)

1. The range for λ_L^* and λ_U^* are determined.

   - If $x_k = l_k$, then $\lambda_L^* \in [\lambda_{min}, \lambda^*]$ with $\mathbf{x}_L^{U*}$

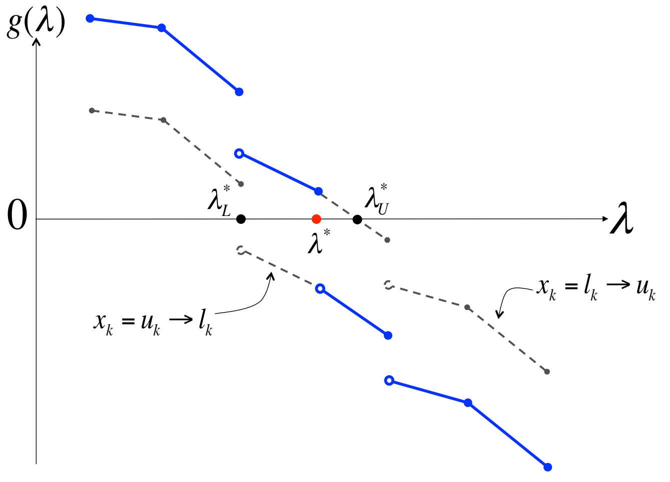


Figure 4.16.: λ_L^* and λ_U^*

- If $x_k = u_k$, then $\lambda_U^* \in [\lambda^*, \lambda_{max}]$ with $\mathbf{x}_U^{U*}$

2. The shape of $g(\lambda)$ are identical to $g(\lambda_L)$ and $g(\lambda_U)$

- If $x_k = l_k$, then $g(\lambda_L) = g(\lambda) - u_k + l_k$
- If $x_k = u_k$, then $g(\lambda_U) = g(\lambda) - l_k + u_k$

Thus, if $g(\lambda)$ is recorded for tested λ's while λ^* is obtained, computations to get λ_L^* and λ_U^* can be obtained fairly less computation than that for λ^*. Once $\mathbf{x}_L^{U*}$, and $\mathbf{x}_U^{U*}$ are obtained, the corresponding objective values are calculated, and one of two solution is selected if its objective value is better than that with $\mathbf{x}^{U*}$. This steps are repeated until $\mathbf{x}^{U*}$ is the best solution or the selected solution does not have k. The preliminary experiments show that this idea actually lead to often find a global optimum solution or improve the solution quality in at most 5 iterations; however, the iteration can be up to $|N|$. Further research on this idea is left for the future research.

5. Applications

This chapter considers applications of convex and nonconvex KSQP. For the convex case, a list of applications is presented with references. As for applications of the nonconvex case, randomly generated application problems are tested with the proposed algorithms and benchmarked for performance.

5.1. Applications for Convex case

Applications for strictly convex KSQP are tremendous and are well collected by the book by Ibaraki and Katoh [54], a survey of Bretthauer and Shetty[16], and Patriksson [106].

Above all, strictly convex KSQP is very useful as a sub-problem in many algorithms as listed below.

KSQP as a sub-problem

- Support vector machine (SVM) by spectral projected gradient method [25, 31, 44]
- Quadratic programming [21, 27, 36, 93, 94, 95, 96]
- Dual ascent algorithm to find the step size [73, 134, 137]
- Discrete nonlinear knapsack problem [67]
- Traffic assignment problem [70, 74]
- Stochastic quasi gradient method [112, 124]

In addiction, the strictly convex KSQP is used mainly for network, traffic, and resource allocation problems. Since Kiwiel [63, 64, 65] and Patriksson [106] gathered many valuable applications, we added more applications based on their work. In addition, applications that are not in their papers are also listed below.

Mostly based on Kiwiel [63, 64, 65]

- Resource allocation [14, 15, 52]
- Hierarchical production planning [14]
- Network flows [134]
- Transportation problems [26]
- Multicommodity network flows [2, 50, 58, 61, 71, 88, 120, 134]
- Integer quadratic knapsack problems [18, 19, 140]
- Integer and continuous quadratic optimization over submodular constraints [52]
- Lagrangian relaxation via subgradient methods [49]

Mostly based on Patriksson [106]

- Decomposition methods for Stochastic programming problems [84, 89, 112]
- Traffic equilibrium problems [11, 29, 28, 70, 74]
- Constrained matrix problems[3, 4, 27, 88, 93, 94, 95, 120, 133, 134]
- Portfolio selection problems [36, 40, 57, 119, 129, 96]

Other applications

- Matrix balancing in regional and national economics [27, 85, 86]
- Economic Dispatch Problem [6, 8, 9]
- Quadratic resource allocation problem with generalized upper bound (GUB): [15]

5.2. Applications for Nonconvex case

Applications for nonconvex KSQP mainly appear in combinatorial optimization problems. The combinatorial book of Du and Pardalos [35] presents a comprehensive list of applications. In this section, we present an application where indefinite KSQP can be used to solve mixed integer problems arising in a portfolio selection problem. Then, subset-sum problem, a direct application of indefinite KSQP, is also discussed with numerical experience.

5.2.1. Real investment portfolio selection

Suppose there exist $N = n + m$ real investment options with uncertain returns. First n options can be invested as a fraction (such as partial ownership), while each of the last m options may only be either fully-invested or not invested at all. Let the vector $\mathbf{x} \in \mathcal{R}^N$ be the investment decision and the mean vector of returns is denoted by $\boldsymbol{\mu} \in \mathcal{R}^N$ if each option is fully-invested. The uncertainty of the investment returns is captured by the variance-covariance matrix $\mathbf{V}$. Then, the risk (variance) of investment returns is given by

$$\mathbf{x}^T\mathbf{V}\mathbf{x}. \tag{5.1}$$

We assume that investments are only allowed to be "long", that is, $x_j \geq 0 \ \forall j$. Then, the decision of the investment can be made with $x_j \in [0,1]$, $j = 1, ..., n$ and $x_j = \{0,1\}$, $j = n+1, ..., N$. Suppose the (unit) cost of investment option j is c_j, $j = 1, ..., N$. The available total budget is denoted by B. Then, the portfolio investment problem is formulated as the mixed-integer QP model in (5.2) that minimizes the risk of investment portfolio, less the invest return, for a given level of the trade-off parameter θ (≥ 0):

$$\begin{aligned} Min \quad & \mathbf{x}^T\mathbf{V}\mathbf{x} - \theta\boldsymbol{\mu}^T\mathbf{x} \\ s.t. \quad & \mathbf{c}^T\mathbf{x} \leq B \\ & x_j \in [0,1], \ j = 1, ..., n \\ & x_j \in \{0,1\}, \ j = n+1, ..., N \end{aligned} \tag{5.2}$$

Pardalos and Rosen [102] and Zhang and Edirisinghe [142] present a technique to transform a given binary mixed-integer QP such as (5.2) to a continuous nonconvex QP by replacing binary constraints with concave continuous functions. Applying this technique, the model in (5.2) can be equivalently written as:

$$\begin{aligned} Min \quad & \mathbf{x}^T\mathbf{V}\mathbf{x} - \theta\boldsymbol{\mu}^T\mathbf{x} + M\textstyle\sum_{j=n+1}^{N} x_j(1-x_j) \\ s.t. \quad & \mathbf{c}^T\mathbf{x} \leq B \\ & x_j \in [0,1],\ j = 1, ..., N \end{aligned} \tag{5.3}$$

where $M \geq 0$ is a penalty parameter to force x_j, $j = n+1, ..., N$ to the binary values.

Consider the special case of investment options that are geographically separated in nature, and thus, their returns are uncorrelated, i.e. $\mathbf{V}$ is a diagonal matrix. Then, (5.3) is an indefinite KSQP that can be efficiently solved by `CBS` and the bounding procedure based on `OBG`, as discussed in Section 3.3.

5.2.1.1. Implementation and computational experiments

Real investment portfolio selection model in (5.2) with uncorrelated invest options is computationally tested in MATLAB with `CBS` and `cplexmiqp.m`, which is a solver of CPLEX 12.6 for mixed integer quadratic problems. CPLEX directly solves (5.2), while `CBS` can only solve (5.3) for a given parameter M.

In order to converge solution of (5.3) to a global solution of (5.2), we iteratively update M, based on the current solution of (5.3). This update is based on a rule that depends on the current M_k, the solution provided by `CBS`, $\mathbf{x}_k$, and the degree of violation v_k of the binary constraints, defined by

$$v_k = \sum_{j=n+1}^{N} x_j^k(1-x_j^k).$$

For a given constant C, this rule is

$$M_{k+1} = M_k(1 + C \cdot max\{1, v_k\}). \tag{5.4}$$

If at some iteration, $v_k = 0$, then $\mathbf{x}^k$ is a binary feasible solution that is approximately global optimum. The implementation of updating rule is presented in Algorithm 5.1. For the experiment, the initial value of $M_0 = 0.1$ is given, and $C = 20$, the maximum iteration $K = 100$, and $\varepsilon_{vio} = 10^{-5}$ are used.

Algorithm 5.1 Mixed Integer programming with `CBS` (`MICBS`)

```
Given that $K$ for maximum number of iteration and $\varepsilon_{vio}$ for tolerance of solution violation
  1. Set $k = 1$, $C = 20$ and $M_0 = 0.1$
  2. Iterate while $k \leq K$
     a) Solve (5.3) using CBS to get $\mathbf{x}^k$
     b) Get the violation value $v_k = \sum_{j=n+1}^{N} x_j(1 - x_j)$
     c) If $|v_k| < \varepsilon_{vio}$, finish algorithm with $\mathbf{x}^k$
     d) Update $M_k = M_{k-1}(1 + C \cdot max\{1, v\})$ and $k \leftarrow k + 1$
```

Experimental results

For the experiment, random problems are generated using the following two steps:

1. Generate random data

   $V_{jj} \in [1, 10]$, $c_j \in [10, 200]$, $\mu_j \in [1\%, 7\%]$, $j = 1, ..., n$, $\mu_j \in [5\%, 15\%]$, $j = n+1, ..., N$

2. Generate a random solution $x_j \in [0, 1]$, $j = 1, ..., n$, $x_j \in \{0, 1\}$, $j = n + 1, ..., N$ to construct $B = \mathbf{c}^T\mathbf{x}$

With a random instance in the size of $N = 1000$ and $n = 500$, the efficient frontier line is drawn in Figure 5.1 with $\theta = 1, 2, ..., 100$. The line is drawn based on the solutions of CPLEX and the dots are plotted using the solutions from `MICBS`. `MICBS` reaches to iteration limit of 100 in three θ's, and the three instances are shown by dots that are considerably far from the efficient frontier line of CPLEX in Figure 5.1. Table 5.2 presents the relative errors of the three instances. The table also shows the 8 instances that `MICBS` finds worse solutions than CPLEX. The instances are selected if the relative error values in (5.5) are greater than

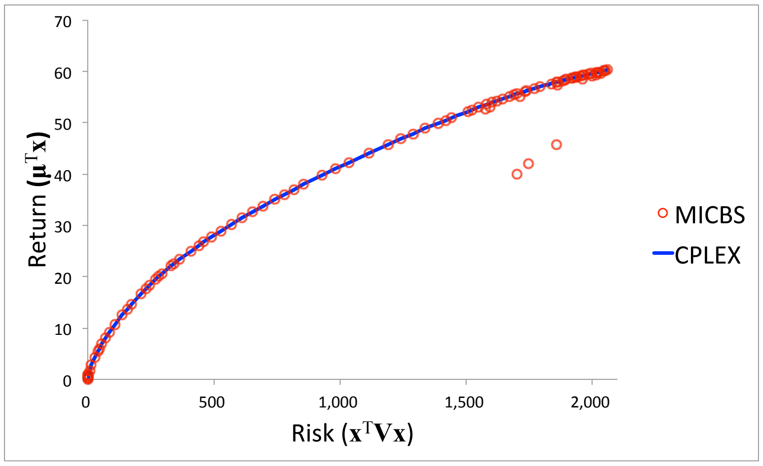


Figure 5.1.: Efficient frontier line

Figure 5.2.: Cases that $Rel.error > 0.001$

	CPU time (sec.)		MICBS			
θ	TimeC	TimeM	Risk	Return	Iteration	Rel.error[1]
66	0.051	0.011	1577.1	52.7	13	3.45%
67	0.021	0.011	1594.1	53.1	12	3.15%
70	0.022	0.006	1714.1	55.1	13	2.49%
76	0.067	0.007	1863.3	57.5	13	1.61%
85	0.019	0.009	1962.3	58.6	11	1.86%
90	0.019	0.009	2001.1	59.2	13	1.42%
92	0.115	0.009	2013.8	59.4	13	1.23%
96	0.019	0.010	2035.8	59.6	13	1.11%
53	0.017	0.137	1859.0	45.8	Max. Iter.[2]	55.80%
71	0.019	0.131	1746.6	41.9	Max. Iter.	45.56%
84	0.078	0.113	1701.2	39.9	Max. Iter.	45.20%

1. $Rel.error = [Obj(\text{MICBS}) - Obj(\text{CPLEX})]/|Obj(\text{CPLEX})|$, where $Obj(\cdot)$ is the objective value in (5.2).
2. Maximum number of iteration is set to 100.

0.001.

$$Rel.error = \frac{Obj(\text{MICBS}) - Obj(\text{CPLEX})}{|Obj(\text{CPLEX})|} \times 100\% \tag{5.5}$$

where $Obj(\cdot)$ is the objective function value in (5.2). The 8 instances have quite small relative errors ranged in [1.11%, 3.45%]. For all other 89 cases, MICBS finds either an exactly same solution as CPLEX or very closely approximate solutions in from 2 to 12 iterations.

The performance of MICBS is also compared with CPLEX in the larger sizes ranged from 5,000 to 100,000. For each size, 10 random problems are generated, and 5 different θ's are used. Table 5.1 presents the average CPU time. MICBS solves all instances in a fraction

Table 5.1.: Average CPU time (in seconds) for real investment portfolio selection

θ	Solvers \ Sizes	5,000	10,000	30,000	50,000	70,000	100,000
10[1]	CPLEX	0.033	0.059	0.294	0.411	0.613	0.928
	MICBS	0.001	0.002	0.004	0.006	0.009	0.014
30	CPLEX	0.038	0.073	0.330	0.470	0.758	1.192
	MICBS	0.001	0.002	0.004	0.007	0.010	0.015
50	CPLEX	0.046	0.095	0.385	0.667	0.920	1.392
	MICBS	0.001	0.002	0.004	0.007	0.010	0.015
70	CPLEX	0.12	0.46	3.90	12.77	28.59	66.18
	MICBS[2]	0.01	0.02	0.02	0.07	0.12	0.17
	Avg. iter.[3] (Max. iter.[4])	5.7	5.5	2.8	4.6 (1)	5.8	5.6
	# *Rel.error* > 0.001[5]	1	0	0	1	0	1
90	CPLEX	0.147	0.560	4.383	14.025	31.414	71.632
	MICBS	0.013	0.014	0.055	0.064	0.129	0.199
	Avg. iter.	7.6	4.6	6.6	4.7	6.4	6.7
	# *Rel.error* > 0.001	1	1	0	1	0	1

1. MICBS finds the same solution as CPLEX in 2 iterations for all instances at θ=10, 30, 50.
2. Average time (in seconds) of MICBS except for instances that reaches iteration limit of 100 (only one case in $N = 50,000$).
3. Average iteration of MICBS except for instances that reaches iteration limit of 100.
4. The number of instances that the iteration limit of 100 reaches in MICBS.
5. The number of instances that $Rel.error = [Obj(\text{MICBS}) - Obj(\text{CPLEX})]/\,|Obj(\text{CPLEX})| \geq 0.001$.

of second for all instances, while CPLEX spends significantly more time than MICBS with higher time growth rates. The quality of solutions of MICBS is excellent. For each problem instance, MICBS has none or one case in which the relative error value in (5.5) is greater than 0.001. Moreover, MICBS reaches the iteration limit in only 1 time over 300 tested problems, yielding the relative error of 39.3% in the case. MICBS also finds either global solution or high quality solution ($Rel.error. < 0.001$) in 2 iterations for all instances at $\theta = 10$, 30, and 50 and in 4 to 7 iterations for instances at larger θ's.

5.2.2. Subset-sum problem

Consider a given set of $1, ..., n$ different items, each of which has resource consumption a_j (> 0). The problem is to determine a subset ($I \subseteq \{1, ..., n\}$) of the items when collected together will not be not too different from a pre-specified capacity b (say of a collector cell). Such problems occur in graph coloring, risk-allocated-capital problems, bin-packing in logistics problems, etc.

The problem is called by subset-sum problem, and it can be mathematically formulated as

$$\begin{aligned} Min_I &\quad \left(b - \textstyle\sum_{i\in I} a_i\right)^2 \\ s.t. &\quad I \subseteq \{1, ..., n\} \end{aligned}$$

to get a subset I. In spite of the simple structure, subset-sum is a well-known *NP-complete* problem. One can refer to Karp (1972, page 380 of [59]) for proof and books of [35, 53, 60, 80] for methods and applications.

Vavasis [132] solves the subset-sum problem in the format of indefinite KSQP:

$$\begin{aligned} Min &\quad y^2 + M \textstyle\sum_{\forall j} x_j(a_j - x_j) \\ s.t. &\quad y + \textstyle\sum_{\forall j} x_j = b \\ &\quad \mathbf{0} \le \mathbf{x} \le \mathbf{a} \\ &\quad y \ free \end{aligned} \tag{5.6}$$

where M is a parameter to control the concavity.

5.2.2.1. Experiments of subset-sum problem

Subset-sum problem is a special case for `CBS` because as mentioned in Section 4.2.2, subset-sum problem is an extreme case that generates $n-1$ identical breakpoints at λ^* for all $j \in N$. In the case, the heuristic method, lower index rule, suggested for `CBS` does not have any theoretical reason to yield a good solution. Thus, `CBS` is not considered for the experiments in subset-sum problem, and only two methods, bounding procedure in Chapter 3 and the local solver of Vavasis [132], are compared. To present the experiment result, as Chapter 3, the upper bound of our algorithm is referred to UB, and a name `VAV92` is used for the algorithm of Vavasis [132].

The random subset-sum problem is generated in the form of (5.6). If the right hand side b is sum of some a_j, then the global objective value of (5.6) is known to be *zero*. For the experiments, a_j are randomly generated in the interval of $(0, round(n/100)]$. Then, the randomly chosen half of a_j are summed up for b. For the concavity parameter, we set $M = 5$.

The two upper bounds of UB and obj(VAV92) are compared using the following metric, named "relative quality of upper bounds" (RQUB):

$$\texttt{RQUB} = \frac{\texttt{obj(VAV92)} - UB}{min\{\texttt{obj(VAV92)}, UB\} + 1}.$$

The large positive value of RQUB means that UB is much better than the upper bound of VAV92 and that UB is closer to the global optimum value of *zero*. Moreover, if RQUB is a negative value close to *zero*, the upper bound of VAV92 is better than UB but the difference is not so great. So when two upper bounds are about to same, RQUB is close to *zero*. Each value of RQUB is plotted (blue circle) in Figure 5.2.2 for 10 instances in each size ranging from $n = 500$ to $5,000$. As observed in the figure, most RQUB values are positive, and many points are far from *zero*. Thus, we conclude UB has quite higher solution quality than VAV92 in the tested problems.

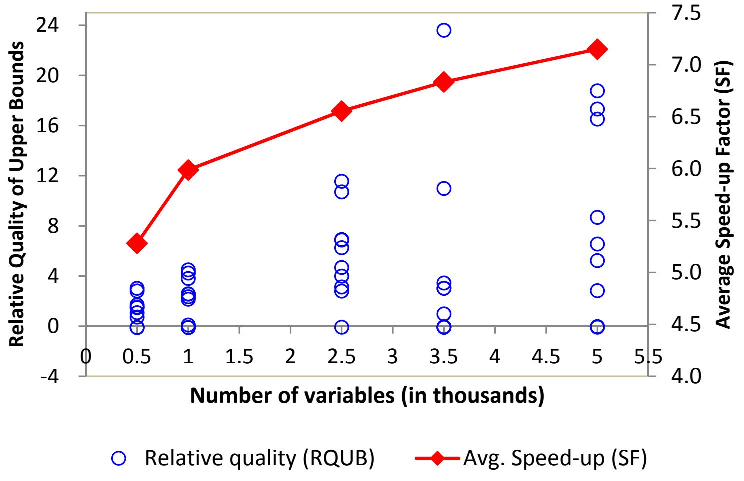


Figure 5.3.: Relative quality of upper bounds (RQUB) on subset-sum problems

The speed is also compared with the speed-up factor (SF), used in section 3.4.2.

$$\texttt{SF} = ln\frac{cpu - time(\texttt{VAV92})}{cpu - time(UB)}$$

It is also drawn in Figure 5.2.2 with a red line, and it indicates that the time growth rate of

Figure 5.4.: Average speed (second) in subset-sum problem

Size n	500	1000	2500	3500	5000
UB of `OBG`	0.004	0.008	0.036	0.061	0.112
`VAV92`	0.734	3.252	25.210	57.068	142.748
`CPX-G`	364.698	1218.367	1236.518	1887.225	1219.483
`CPX-G` time range	[1.9, 777.9]	[3.0, 3642.0]	[28.3, 3915.6]	[103.6, 4886]	[61.7, 4654.1]

UB is quite less than `VAV92` in a higher speed. The average speeds of each method are also available in Table 5.4. It shows that the upper bounding method using `OBG` is also superior in the speed as well as the quality.

6. Concluding Remarks

This dissertation focuses on the goal: "*developing efficient algorithms to solve general KSQP*". Toward this goal, three new methods for nonconvex case are developed and applications for both convex and nonconvex are discussed.

Contributions

The goal of this dissertation is achieved with the following five main contributions.

First, the computational efficiency and theoretical complexity of existing methods are discussed through the comprehensive literature review for convex and nonconvex KSQP. In addition, several ideas that improve the computational complexity and efficiency of the existing methods are suggested and utilized for the development of our new algorithms.

Second, a new global algorithm, `OBG`, that solves the open-box constrained indefinite KSQP in $O(n^2)$ is developed. The global optimality is guaranteed by choosing the best solution among all enumerated KKT points that satisfy additional necessary conditions. In spite of the enumeration steps, the $O(n^2)$ time complexity is achieved by utilizing techniques in the interval test method and by developing Lagrange multiplier domain shrinking procedure. The superior performance is verified through the experimental comparison with local, global, and commercial solvers.

Third, a new bounding algorithm for general KSQP, which has closed-box constraints, is proposed utilizing the global optimizer `OBG`. A lower bound is obtained by searching for Lagrangian multipliers, and an upper bound is found based on the lower bound solution and KKT conditions. In computational experiments using very large size problems, the bounds are found to be quite tight, as well as computations are relatively faster in comparison to other tested solvers.

Fourth, another new algorithm, `CBS`, is developed to find lower and upper bounds for general KSQP. The gap between two bounds is expected to be tight because it is determined by the given coefficients of a single variable, and the conditions to ensure the global optimality of the solution are also presented. The algorithm inherits efficiency and reliability due to the newly developed hybrid method that combines techniques in strictly convex KSQP. `CBS` and other existing solvers are compared, and experiment results show the superiority of `CBS` in speed as well as in solution quality.

Fifth, an extensive list of applications is presented for the convex KSQP. A practical investment portfolio selection problem is used to show that `CBS` can be used to solve mixed integer problems very efficiently, and the performance of the upper bounding procedure, which utilizes `OBG`, is compared with a local solver using the subset-sum problem as the prototype of various applications.

Future directions of research

KSQP is an important class for research with regard to efficient algorithmic development in nonlinear programming as well as in complexity theory.

Quadratic Programming Indefinite KSQP is useful for its direct applications in combinatorial problems such as subset-sum problem, and furthermore, its potential value is found in mixed integer programming as we demonstrated with the portfolio selection model in Section 5.2.1. The considered model has a separable objective function and only a knapsack constraint with box constraints, but it can be extended to more general problems. For example, multiple-constrained quadratic integer problems are considered in Edirisinghe [37] applying the transformation in Zhang and Edirisinghe [142] supplemented with constraint-relaxation. The efficient solution technique developed in this dissertation for the single constraint case can be incorporated in the multiple-constrained integer problems to develop a new class of solution methodologies. Moreover, as discussed in the introduction chapter, the methods developed for indefinite KSQP in this dissertation can be used as a fundamental building-block to solve general QPs such as nonseparable QP and indefinite QP when the row-aggregation and diagonalization techniques are applied. Thus, developing algorithms toward various QP utilizing the proposed methods will be a part of future research.

NP-hard problem Indefinite KSQP is one of the simple form of *NP-hard* problems, and it is involved in a long time unproved theoretic question *"P=NP?"*. This question may be able to be explained through the `CBS` method. As illustrated in Figure 6.1, `CBS` finds an exact global optimum solution when the right hand side b is not in the range of

$$B = \bigcup_{k \in N} \left(\sum_{\forall j} \hat{x}_j(t_k), \ \sum_{\forall j} x_j(t_k) \right)$$

where t_j in (4.9), $\hat{x}_j(\lambda)$ in (4.15), $x_j(\lambda)$ in (4.2).

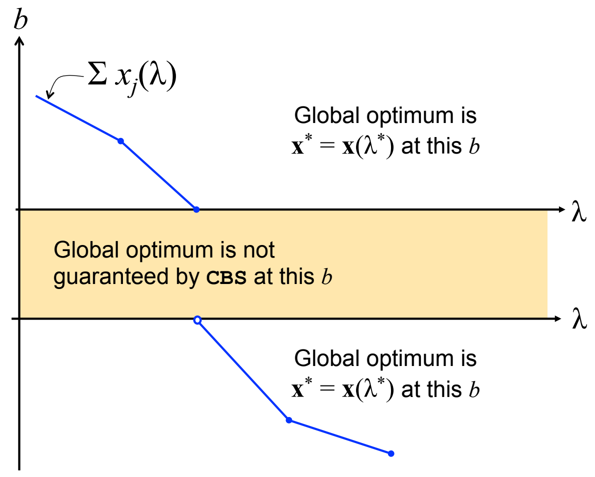


Figure 6.1.: Global optimality depending on RHS b

According to the geometric interpretation of interval test in Section 2.2.3, the interval or singleton partitions constructed by breakpoints can be geometrically interpreted as a face, edge, vertex, or inside of the box (constraints) in the convex case. However, in the indefinite case, the partitions, which are constructed at t_j, represent multiple faces of the box since x_j, $j \in N$ can be one of three values l_j, u_j, and $-\lambda/d_j$. Further research is required to explore the properties of the partition at t_j.

Improvement of proposed methods Experimental results show that the methods developed in this dissertation have superior performance in speed and solution quality. But some ideas that can further improve the proposed methods are still remained for the future research. For example, in the lower bounding procedure using `OBG` in Section 3.3, the method integrates multiple Lagrange multipliers into the univariate multiplier w because updating $|N|$ number of multipliers requires a more sophisticated procedure; however, the surrogation technique [48, 87], which has been studied since 1970's, may be an candidate for the multiplier updating procedure.

In Section 4.5, three ideas to improve `CBS` are discussed. The third idea that minimizes the *bound gap* has been already tested, and the results are highly promising. So the immediate research on the idea will be followed after this dissertation.

Reference

[1] Absil, P. A., and Tits, A. L. Newton-kkt interior-point methods for indefinite quadratic programming. *Computational Optimization and Applications 36*, 1 (2007), 5–41.

[2] Ali, A., Helgason, R., Kennington, J., and Lall, H. Computational comparison among three multicommodity network flow algorithms. *Operations Research 28*, 4 (1980), 995–1000.

[3] Bachem, A., and Korte, B. An algorithm for quadratic optimization over transportation polytopes. *Zeitschrift für Angewandte Mathematik und Mechanik 58* (1978), 459–461.

[4] Bachem, A., and Korte, B. Minimum norm problems over transportation polytopes. *Linear Algebra and its Applications 31* (1980), 103–118.

[5] Barrientos, O., and Correa, R. An algorithm for global minimization of linearly constrained quadratic functions*. *Journal of Global Optimization 16*, 1 (2000), 77–93.

[6] Bayón, L., Grau, J., Ruiz, M., and Suarez, P. New developments in the application of pontryagin's principle for the hydrothermal optimization. *IMA Journal of Mathematical Control and Information 22*, 4 (2005), 377–393.

[7] Bayón, L., Grau, J., Ruiz, M., and Suárez, P. A quasi-linear algorithm for calculating the infimal convolution of convex quadratic functions. *Journal of Computational and Applied Mathematics 236*, 12 (2012), 2990–2997.

[8] Bayón, L., Grau, J., Ruiz, M., and Suárez, P. An exact algorithm for the continuous quadratic knapsack problem via infimal convolution. In *Handbook of Optimization*. Springer, 2013, pp. 97–127.

[9] Bayón, L., Grau, J., and Suárez, P. A new formulation of the equivalent thermal in optimization of hydrothermal systems. *Mathematical Problems in Engineering 8*, 3 (2002), 181–196.

[10] Bazaraa, M. S., Sherali, H. D., Shetty, C., et al. Nonlinear programming, theory and applications, 1993.

[11] Bertsekas, D. P., and Gafni, E. M. *Projection methods for variational inequalities with application to the traffic assignment problem.* Springer, 1982.

[12] Birgin, E. G., Martínez, J. M., and Raydan, M. Nonmonotone spectral projected gradient methods on convex sets. *SIAM Journal on Optimization 10*, 4 (2000), 1196–1211.

[13] Birgin, E. G., Martínez, J. M., and Raydan, M. Algorithm 813: Spg—software for convex-constrained optimization. *ACM Transactions on Mathematical Software (TOMS) 27*, 3 (2001), 340–349.

[14] Bitran, G. R., and Hax, A. C. Disaggregation and resource allocation using convex knapsack problems with bounded variables. *Management Science 27*, 4 (1981), 431–441.

[15] Bretthauer, K. M., and Shetty, B. Quadratic resource allocation with generalized upper bounds. *Operations Research Letters 20*, 2 (1997), 51 – 57.

[16] Bretthauer, K. M., and Shetty, B. The nonlinear knapsack problem–algorithms and applications. *European Journal of Operational Research 138*, 3 (2002), 459–472.

[17] Bretthauer, K. M., and Shetty, B. A pegging algorithm for the nonlinear resource allocation problem. *Computers & Operations Research 29*, 5 (2002), 505 – 527.

[18] Bretthauer, K. M., Shetty, B., and Syam, S. A branch and bound algorithm for integer quadratic knapsack problems. *ORSA Journal on Computing 7*, 1 (1995), 109–116.

[19] Bretthauer, K. M., Shetty, B., and Syam, S. A projection method for the integer quadratic knapsack problem. *Journal of the Operational Research Society* (1996), 457–462.

[20] Brucker, P. An o(n) algorithm for quadratic knapsack problems. *Operations Research Letters 3*, 3 (1984), 163 – 166.

[21] Calamai, P. H., and Moré, J. J. Quasi-newton updates with bounds. *SIAM journal on numerical analysis 24*, 6 (1987), 1434–1441.

[22] Chazelle, B., and Guibas, L. J. Visibility and intersection problems in plane geometry. *Discrete & Computational Geometry 4*, 1 (1989), 551–581.

[23] Chen, J., and Burer, S. Globally solving nonconvex quadratic programming problems via completely positive programming. *Mathematical Programming Computation 4*, 1 (2012), 33–52.

[24] Chinchuluun, A., Pardalos, P. M., and Enkhbat, R. Global minimization algorithms for concave quadratic programming problems. *Optimization 54*, 6 (2005), 627–639.

[25] Cominetti, R., Mascarenhas, W. F., and Silva, P. J. A newton's method for the continuous quadratic knapsack problem. *Available: http://www.optimization-online.org/* (2012).

[26] Cosares, S., and Hochbaum, D. S. Strongly polynomial algorithms for the quadratic transportation problem with a fixed number of sources. *Mathematics of Operations Research 19*, 1 (1994), 94–111.

[27] Cottle, R. W., Duvall, S. G., and Zikan, K. A lagrangean relaxation algorithm for the constrained matrix problem. *Naval Research Logistics Quarterly 33*, 1 (1986), 55–76.

[28] Dafermos, S., and Nagurney, A. Supply and demand equilibration algorithms for a class of market equilibrium problems. *Transportation Science 23*, 2 (1989), 118–124.

[29] Dafermos, S. C., and Sparrow, F. T. The traffic assignment problem for a general network. *Journal of Research of the National Bureau of Standards, Series B 73*, 2 (1969), 91–118.

[30] Dai, Y.-H., and Fletcher, R. Projected barzilai-borwein methods for large-scale box-constrained quadratic programming. *Numerische Mathematik 100*, 1 (2005), 21–47.

[31] Dai, Y.-H., and Fletcher, R. New algorithms for singly linearly constrained quadratic programs subject to lower and upper bounds. *Mathematical Programming 106*, 3 (2006), 403–421.

[32] De Waegenaere, A., and Wielhouwer, J. A breakpoint search approach for convex resource allocation problems with bounded variables. *Optimization Letters 6* (2012), 629–640.

[33] Dolan, E. D., and Moré, J. J. Benchmarking optimization software with performance profiles. *Mathematical programming 91*, 2 (2002), 201–213.

[34] Dostál, Z. *Optimal quadratic programming algorithms: with applications to variational inequalities*, vol. 23. Springer, 2009.

[35] Du, D.-Z., and Pardalos, P. M. *Handbook of combinatorial optimization*, vol. 3. Springer, 1998.

[36] Dussault, J. P., Ferland, J. A., and Lemaire, B. Convex quadratic programming with one constraint and bounded variables. *Mathematical Programming 36*, 1 (1986), 90–104.

[37] Edirisinghe, N. Block separable optimization via quadratic decomposition: Case of binary integer programs. *Working paper, University of Tennessee* (2010).

[38] Edirisinghe, N. Index-tracking optimal portfolio selection. *Quantitative Finance Letters*, 1 (2013), 16–20.

[39] Einbu, J. M. Extension of the luss-gupta resource allocation algorithm by means of first order approximation techniques. *Operations Research 29*, 3 (1981), 621–626.

[40] Elton, E. J., Gruber, M. J., and Padberg, M. W. Simple criteria for optimal portfolio selection. *The Journal of Finance 31*, 5 (1976), 1341–1357.

[41] Floudas, C., and Pardalos, P. Recent advances in global optimization. Tech. rep., DTIC Document, 1992.

[42] Floudas, C. A., and Visweswaran, V. Quadratic optimization. In *Handbook of Global Optimization* (Dordrecht, The Netherlands, 1995), R. Horst and P. M. Pardalos, Eds., Kluwer Academic Publishers.

[43] Floyd, R. W., and Rivest, R. L. Algorithm 489: the algorithm select for finding the i th smallest of n elements. *Communications of the ACM 18*, 3 (1975), 173.

[44] Fu, Y.-S., and Dai, Y.-H. Improved projected gradient algorithms for singly linearly constrained quadratic programs subject to lower and upper bounds. *APJOR 27*, 1 (2010), 71–84.

[45] Gill, P., Murray, W., and Wright, M. Numerical linear algebra and optimization. *Addison-Wesley* (1991).

[46] Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H. Procedures for optimization problems with a mixture of bounds and general linear constraints. *ACM Transactions on Mathematical Software (TOMS) 10*, 3 (1984), 282–298.

[47] Gill, P. E., Murray, W., and Wright, M. H. Practical optimization.

[48] Greenberg, H. J., and Pierskalla, W. P. Surrogate mathematical programming. *Operations Research 18*, 5 (1970), 924–939.

[49] Held, M., Wolfe, P., and Crowder, H. Validation of subgradient optimization. *Mathematical Programming 6* (1974), 62–88.

[50] Helgason, R., Kennington, J. L., and Lall, H. A polynomially bounded algorithm for a singly constrained quadratic program. *Mathematical Programming 18*, 3 (1980), 338–343.

[51] Hoare, C. A. Quicksort. *The Computer Journal 5*, 1 (1962), 10–16.

[52] Hochbaum, D. S., and Hong, S. P. About strongly polynomial-time algorithms for quadratic optimization over submodular constraints. *Mathematical Programming 69*, 2 (1995), 269–309.

[53] Horst, R., Pardalos, P. M., and Van Thoai, N. *Introduction to global optimization.* Kluwer Academic Pub, 2000.

[54] Ibaraki, T., and Katoh, N. *Resource allocation problems: algorithmic approaches.* MIT press, 1988.

[55] Illés, T., and Nagy, A. Sufficient optimality criterion for linearly constrained, separable concave minimization problems. *journal of optimization theory and applications 125*, 3 (2005), 559–575.

[56] ISO. Iso/iec 14882: 2003 programming languages c++. Tech. rep., Technical report, ISO, 2003.

[57] Jucker, J. V., and de Faro, C. A simple algorithm for stone's version of the portfolio selection problem. *Journal of Financial and Quantitative Analysis* (1975), 859–870.

[58] Kamesam, P., and Meyer, R. Multipoint methods for separable nonlinear networks. In *Mathematical Programming at Oberwolfach II.* Springer, 1984, pp. 185–205.

[59] Karp, R. M. *Reducibility among combinatorial problems.* Springer, 1972.

[60] Kellerer, H., Pferschy, U., and Pisinger, D. *Knapsack problems.* Springer Verlag, 2004.

[61] Kennington, J., and Shalaby, M. An effective subgradient procedure for minimal cost multicommodity flow problems. *Management Science 23*, 9 (1977), 994–1004.

[62] Kim, G., and Wu, C.-H. A pegging algorithm for separable continuous nonlinear knapsack problems with box constraints. *Engineering Optimization 44*, 10 (2012), 1245–1259.

[63] Kiwiel, K. On linear-time algorithms for the continuous quadratic knapsack problem. *Journal of Optimization Theory and Applications 134*, 3 (2007), 549–554.

[64] Kiwiel, K. Breakpoint searching algorithms for the continuous quadratic knapsack problem. *Mathematical Programming 112* (2008), 473–491.

[65] Kiwiel, K. Variable fixing algorithms for the continuous quadratic knapsack problem. *Journal of Optimization Theory and Applications 136*, 3 (2008), 445–458.

[66] Kiwiel, K. C. On floyd and rivest's select algorithm. *Theoretical Computer Science 347* (2005), 214 – 238.

[67] Klastorin, T. On a discrete nonlinear and nonseparable knapsack problem. *Operations Research Letters 9*, 4 (1990), 233–237.

[68] Kough, P. F. Global solution to the indefinite quadratic programming problem. Tech. rep., Washington Univ., St Louis, MO, USA, 1974.

[69] Kough, P. F. The indefinite quadratic programming problem. *Operations Research 27*, 3 (1979), 516–533.

[70] Larsson, T., and Patriksson, M. Simplicial decomposition with disaggregated representation for the traffic assignment problem. *Transportation Science 26*, 1 (1992), 4–17.

[71] Larsson, T., Patriksson, M., and Strömberg, A.-B. Conditional subgradient optimizationtheory and applications. *European Journal of Operational Research 88*, 2 (1996), 382–403.

[72] Lin, C., Lucidi, S., Palagi, L., Risi, A., and Sciandrone, M. Decomposition algorithm model for singly linearly-constrained problems subject to lower and upper bounds. *Journal of Optimization Theory and Applications 141* (2009), 107–126.

[73] Lin, Y., and Pang, J. Iterative methods for large convex quadratic programs: a survey. *SIAM Journal on Control and Optimization 25* (1987), 383–411.

[74] Lotito, P. A. Issues in the implementation of the dsd algorithm for the traffic assignment problem. *European journal of operational research 175*, 3 (2006), 1577–1587.

[75] Luss, H., and Gupta, S. K. Technical noteallocation of effort resources among competing activities. *Operations Research 23*, 2 (1975), 360–366.

[76] Maculan, N., and de Paula, G. G. A linear-time median-finding algorithm for projecting a vector on the simplex of rn. *Operations research letters 8*, 4 (1989), 219–222.

[77] Maculan, N., Minoux, M., and Plateau, G. A o(t) algorithm for projecting a vector on the intersection of a hyperplane and rn+. *RAIRO. Recherche opérationnelle 31*, 1 (1997), 7–16.

[78] Maculan, N., Santiago, C., Macambira, E. M., and Jardim, M. An o (n) algorithm for projecting a vector on the intersection of a hyperplane and a box in r n. *Journal of optimization theory and applications 117*, 3 (2003), 553–574.

[79] Martello, S., Pisinger, D., and Toth, P. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science 45*, 3 (1999), 414–424.

[80] Martello, S., and Toth, P. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[81] MATHWORKS. Accelerating matlab. *Technology Backgrounder (an Technology Backgrounder (an internet newsletter)* (2002).

[82] Michelot, C. A finite algorithm for finding the projection of a point onto the canonical simplex of n. *Journal of Optimization Theory and Applications 50*, 1 (1986), 195–200.

[83] Moré, J. J., and Vavasis, S. A. On the solution of concave knapsack problems. *Mathematical programming 49*, 1 (1991), 397–411.

[84] Mulvey, J. M., and Vladimirou, H. Solving multistage stochastic networks: An application of scenario aggregation. *Networks 21*, 6 (1991), 619–643.

[85] Nagurney, A. An algorithm for the solution of a quadratic-programming problem, with application to constrained matrix and spatial price equilibrium problems. *Environment and planning A 21*, 1 (1989), 99–114.

[86] Nagurney, A., Kim, D.-S., and Robinson, A. G. Serial and parallel equilibration of large-scale constrained matrix problems with application to the social and economic sciences. *International Journal of High Performance Computing Applications 4*, 1 (1990), 49–71.

[87] Nakagawa, Y., James, R. J., Rego, C., and Edirisinghe, N. Entropy-based optimization of nonlinear separable discrete decision models. *Management Science* (to appear in 2014).

[88] Nielsen, S. S., and Zenios, S. A. Massively parallel algorithms for singly constrained convex programs. *ORSA Journal on Computing 4*, 2 (1992), 166–181.

[89] Nielsen, S. S., and Zenios, S. A. A massively parallel algorithm for nonlinear stochastic network problems. *Operations Research 41*, 2 (1993), 319–337.

[90] Nowak, I. Some heuristics and test problems for nonconvex quadratic programming over a simplex. Preprint series, Institut für Mathematik, Humboldt-Universität zu Berlin, Germany, 1998.

[91] Nowak, I. A global optimality criterion for nonconvex quadratic programming over a simplex. Preprint series, Institut für Mathematik, Humboldt-Universität zu Berlin, Germany, 1998b.

[92] Nowak, I. A new semidefinite programming bound for indefinite quadratic forms over a simplex. *Journal of Global Optimization 14*, 4 (1999), 357–364.

[93] Ohuchi, A., and Kaji, I. Algorithms for optimal allocation problems having quadratic objective function. *Journal of the Operations Research Society of Japan 23*, 1 (1980), 64–79.

[94] Ohuchi, A., and Kaji, I. An algorithm for the Hitchcock transportation problems with quadratic cost functions. *Journal of the Operations Research Society of Japan 24*, 2 (1981), 170–181.

[95] Ohuchi, A., and Kaji, I. Lagrangian dual coordinatewise maximization algorithm for network transportation problems with quadratic costs. *Networks 14*, 4 (1984), 515–530.

[96] Pang, J.-S. A new and efficient algorithm for a class of portfolio selection problems. *Operations Research 28*, 3-Part-II (1980), 754–767.

[97] Pardalos, P., and Rosen, J. Methods for global concave minimization: A bibliographic survey. *Siam Review 28*, 3 (1986), 367–379.

[98] Pardalos, P. M. Quadratic problems defined on a convex hull of points. *BIT 28* (1988), 323–328.

[99] Pardalos, P. M. Global optimization algorithms for linearly constrained indefinite quadratic problems. *Computational Mathematics and Applications 21*, 6/7 (1991), 87–97.

[100] Pardalos, P. M. On the passage from local to global in optimization. *Mathematical Programming: State of the Art* (1994), 220–247.

[101] Pardalos, P. M., and Kovoor, N. An algorithm for a singly constrained class of quadratic programs subject to upper and lower bounds. *Mathematical Programming 46*, 3 (1990), 321–328.

[102] Pardalos, P. M., and Rosen, J. B. *Constrained global optimization: algorithms and applications*. Springer-Verlag New York, Inc., 1987.

[103] Pardalos, P. M., and Schnitger, G. Checking local optimality in constrained quadratic programming is NP-hard. *Operations Research Letters 7*, 1 (1988), 33–35.

[104] Pardalos, P. M., and Vavasis, S. A. Quadratic programming with one negative eigenvalue is NP-hard. *Journal of Global Optimization 1* (1991), 15–23.

[105] Pardalos, P. M., Ye, Y., and Han, C.-G. Algorithms for the solution of quadratic knapsack problems. *Linear Algebra and its Applications 152* (1991), 69–91.

[106] Patriksson, M. A survey on the continuous nonlinear resource allocation problem. *European Journal of Operational Research 185*, 1 (2008), 1 – 46.

[107] Phong, T. Q., Hoai An, L. T., and Tao, P. D. Decomposition branch and bound method for globally solving linearly constrained indefinite quadratic minimization problems. *Operations Research Letters 17*, 5 (1995), 215–220.

[108] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[109] Quarteroni, A., Sacco, R., and Saleri, F. *Numerical mathematics*, vol. 37. Springer, 2006.

[110] Riciniello, F. A survey on the methods of linear and quadratic programming. *Note Recensioni e Notizie 22*, 2–3 (1973), 270–315.

[111] Robinson, A. G., Jiang, N., and Lerme, C. S. On the continuous quadratic knapsack-problem. *Mathematical Programming 55*, 1 (1992), 99–108.

[112] Rockafellar, R., and Wets, R. *A note about projections in the implementation of stochastic quasigradient methods*, vol. 10. Numerical Techniques for Stochastic Optimization (Yu. Ermoliev and RJ-B. Wets, eds.), Springer Ser. Comput. Math, 1988.

[113] Romeijn, H. E., Geunes, J., and Taaffe, K. On a nonseparable convex maximization problem with continuous knapsack constraints. *Operations research letters 35*, 2 (2007), 172–180.

[114] Rosen, J. B., and Pardalos, P. M. Global minimization of large-scale constrained concave quadratic problems by separable programming. *Mathematical Programming 34*, 2 (1986), 163–174.

[115] Sahni, S. Computationally related problems. *SIAM Journal on Computing 3*, 4 (1974), 262–279.

[116] Sanathanan, L. On an allocation problem with multistage constraints. *Operations Research 19*, 7 (1971), 1647–1663.

[117] Shampine, L., and Watts, H. Subroutine fzero. *Numerical Methods and Software* (1988).

[118] Sharkey, T., Romeijn, H., and Geunes, J. A class of nonlinear nonseparable continuous knapsack and multiple-choice knapsack problems. *Mathematical Programming 126* (2011), 69–96.

[119] Sharpe, W. F. A simplified model for portfolio analysis. *Management science 9*, 2 (1963), 277–293.

[120] Shetty, B., and Muthukrishnan, R. A parallel projection for the multicommodity network model. *Journal of the Operational Research Society* (1990), 837–842.

[121] Shor, N. Z., Kiwiel, K. C., and Ruszczynski, A. *Minimization methods for non-differentiable functions*, vol. 3. Springer-Verlag Berlin, 1985.

[122] Smith, S., and Lasdon, L. Solving large sparse nonlinear programs using grg. *ORSA Journal on Computing 4*, 1 (1992), 2–15.

[123] Srikantan, K. A problem in optimum allocation. *Operations Research 11*, 2 (1963), 265–273.

[124] Stefanov, S. M. On the implementation of stochastic quasigradient methods to some facility location problems. *The Yugoslav Journal of Operations Research ISSN: 0354-0243 EISSN: 2334-6043 10*, 2 (2000).

[125] Stefanov, S. M. Convex separable minimization subject to bounded variables. *Computational Optimization and Applications 18*, 1 (2001), 27–48.

[126] Stefanov, S. M. *Separable programming: theory and methods*, vol. 53. Kluwer Academic Pub, 2001.

[127] Stefanov, S. M. Convex quadratic minimization subject to a linear constraint and box constraints. *Applied Mathematics Research eXpress 2004*, 1 (2004), 17–42.

[128] Stefanov, S. M. Polynomial algorithms for projecting a point onto a region defined by a linear constraint and box constraints in n. *Journal of Applied Mathematics 2004*, 5 (2004), 409–431.

[129] Stone, B. K. A linear programming formulation of the general portfolio selection problem. *Journal of Financial and Quantitative Analysis 8*, 04 (1973), 621–636.

[130] Tseng, P. Dual ascent methods for problems with strictly convex costs and linear constraints: A unified approach. *SIAM Journal on Control and Optimization 28*, 1 (1990), 214–242.

[131] Vavasis, S. A. Approximation algorithms for indefinite quadratic programming. *Mathematical Programming 57*, 2 (1992), 279–311.

[132] Vavasis, S. A. Local minima for indefinite quadratic knapsack-problems. *Mathematical Programming 54*, 2 (1992), 127–153.

[133] Ventura, J. A. Algorithms for quadratic transportation networks. In *Decision and Control, 1989., Proceedings of the 28th IEEE Conference on* (1989), IEEE, pp. 1131–1135.

[134] Ventura, J. A. Computational development of a lagrangian dual approach for quadratic networks. *Networks 21*, 4 (1991), 469–485.

[135] Washburn, A. R. Technical note on constrained maximization of a sum. *Operations Research 29*, 2 (1981), 411–414.

[136] Wright, S. E., and Rohal, J. J. Solving the continuous nonlinear resource allocation problem with an interior point method. *arXiv preprint arXiv:1305.1284* (2013).

[137] Wu, C.-H., Ventura, J. A., and Browning, S. Computational comparisons of dual conjugate gradient algorithms for strictly convex networks. *Computers & operations research 25*, 4 (1998), 333–349.

[138] Xue, H.-g., Xu, C.-x., and Xu, F.-m. A branch and bound algorithm for separable concave programming. *Journal of Computational Mathematics 22*, 6 (2004), 895–904.

[139] Yajima, Y. Quadratic knapsack. In *Encyclopedia of Optimization, Second Edition*, C. A. Floudas and P. M. Pardalos, Eds. Springer, 2009, pp. 3159–3161.

[140] Zhang, B., and Hua, Z. Simple solution methods for separable mixed linear and quadratic knapsack problem. *Applied Mathematical Modelling 36*, 7 (2012), 3245 – 3256.

[141] Zhang, X. The generalized dea model of fundamental analysis of public firms, with application to portfolio selection. *PhD dissertation, University of Tennessee* (2007).

[142] Zhang, X., and Edirisinghe, N. Optimality conditions and solution methodology for parameter selection in dea. *International Journal of Mathematics in Operational Research 3*, 3 (2011), 245–263.

[143] Zipkin, P. H. Simple ranking methods for allocation of one resource. *Management Science 26*, 1 (1980), 34–43.

Appendix

A. Appendix

A.1. The case of $a_j = 0$

For the variables that $a_j = 0$ in (P), the optimal solution x_j^* can be obtained by

$$x_j^* = \begin{cases} \begin{cases} median\{l_j, u_j, c_j/d_j\} \end{cases} & for\ d_j > 0 \\ \begin{cases} l_j & if\ c_j < d_j(l_j + u_j)/2 \\ u_j & otherwise \end{cases} & for\ d_j < 0 \\ \begin{cases} l_j & if\ c_j < 0 \\ u_j & otherwise \end{cases} & for\ d_j = 0 \end{cases}$$

Note that the case that $c_j = 0$ and $d_j = 0$ does not exist.

A.2. Transformation

Any problem in the format of (P) can be equivalently transformed to a problem such that

$$a_j = 1\ \forall j \qquad \text{and} \qquad c_j = 0\ for\ d_j \neq 0$$

by substituting x_j with

$$x_j = \begin{cases} \hat{x}_j/a_j + c_j/d_j & for\ d_j \neq 0 \\ \hat{x}_j/a_j & for\ d_j = 0 \end{cases}$$

Then, coefficients are as below

$$\hat{d}_j = d_j/a_j^2 \; for \; d_j \neq 0, \qquad \hat{b} = b - \textstyle\sum_{d_j \neq 0} a_j c_j/d_j$$

$$\hat{l}_j = \begin{cases} a_j(l_j - c_j/d_j) & if \; a_j > 0 \\ a_j(u_j - c_j/d_j) & if \; a_j < 0 \end{cases} for \; d_j \neq 0, \qquad \hat{l}_j = \begin{cases} a_j l_j & if \; a_j > 0 \\ a_j u_j & if \; a_j < 0 \end{cases} for \; d_j = 0$$

$$\hat{u}_j = \begin{cases} a_j(u_j - c_j/d_j) & if \; a_j > 0 \\ a_j(l_j - c_j/d_j) & if \; a_j < 0 \end{cases} for \; d_j \neq 0, \qquad \hat{u}_j = \begin{cases} a_j u_j & if \; a_j > 0 \\ a_j l_j & if \; a_j < 0 \end{cases} for \; d_j = 0$$

$$\hat{c}_j = \begin{cases} 0 & for \; d_j \neq 0 \\ c_j/a_j & for \; d_j = 0 \end{cases}$$

The objective value of the original problem can be obtained from the solution $\hat{\mathbf{x}}$ of the transformed problem by

$$Obj = \frac{1}{2}\left(\hat{\mathbf{x}}'\hat{\mathbf{D}}\hat{\mathbf{x}} - \mathbf{c}'\bar{\mathbf{D}}\mathbf{c}\right) - \sum_{d_j=0} \hat{c}_j \hat{x}_j$$

where $\bar{\mathbf{D}}$ is a partial inverse of $\mathbf{D}$ depending on its diagonal elements d_j by

$$\bar{d}_j = \begin{cases} 1/d_j & for \; d_j \neq 0 \\ 0 & for \; d_j = 0 \end{cases}.$$

A.3. Transformation computation

Computational cost of the transformed and the original problems are available in the Table A.1 for the worst case that $Z = \emptyset$ with $|P| = n_1$, $|N| = n_2$ where $|\cdot|$ is the number of elements of the corresponding vector so $n_1 + n_2 = n$. According to Table A.1, the transformed problem is more efficient than the original problem if the difference is less than

Table A.1.: Computational comparison of transformed and original problems

	With Transformation		Without Transformation	
	Computation	Cost	Computation	Cost
Objective value	$\sum_{\forall j} d_j x_j^2$	$3n-1$	$\sum_{\forall j} d_j x_j^2 - c_j x_j$	$5n-1$
Break points	$-d_j u_j$ and $-d_j l_j$ $-d_j(l_j+u_j)/2$	$2n_1$†1 $3n_2$†2	$(c_j - d_j l_j)/a_j$ and $(c_j - d_j u_j)/a_j$ $\left[c_j - \frac{1}{2}d_j(l_j+u_j)\right]/a_j$	$6n_1$†1 $5n_2$†2
Transfo-rmation†3	Including back transformation for $\mathbf{x}$	$15n-1$		
Total (fixed)	$18n-2+2n_1+3n_2$		$5n-1+6n_1+5n_2$	
$\mathbf{x}(\lambda)$ in an iteration	$median\{l_j, u_j, -\lambda/d_j\}$ $\lambda > breakpoint$	$3n_1$ n_2	$median\{l_j, u_j,$ $(c_j - \lambda a_j)/d_j\}$ $\lambda > breakpoint$	$5n_1$ n_2
$g(\lambda)$ in an iteration	$\sum_{\forall j} x_j(\lambda) - b$	n	$\mathbf{a}^T\mathbf{x}(\lambda) - b$	$2n$
Total (in an iteration)	$3n_1+n_2+n$		$5n_1+n_2+2n$	

Note that we do not count operations for changing the sign.
†1 Operations to compute breakpoint for $d_j > 0$
†2 Operations to compute breakpoint for $d_j \leq 0$

zero as

$$(13n - 2 - 4n_1 - 2n_2) + iteration \times (-2n_1 - n) < 0,$$

and the number of iteration that solving the transformed problem is more efficient than the original problem is obtained by considering tow extreme cases: strictly convex and strictly concave cases.

If (P) is in the strictly convex case (that is, $n_1 = n$ and $n_2 = 0$),

$$Iteration \geq 3 > 3 - \frac{2}{3n}.$$

If (P) is in the strictly concave case (that is, $n_1 = 0$ and $n_2 = n$),

$$Iteration \geq 11 > 11 - \frac{2}{n}.$$

Thus, the transformed problem is computationally more efficient than the original problem if the iteration is more than 3 for the strictly convex case and more than 11 for any general case.

A.4. Another derivation of $x_j(\lambda)$ for $j \in N$

In addition to the geometric intuition in the Section 4.2.1 for x_j $j \in N$, we can find it algebraically comparing the objective values.

- If $x_j = l_j$ results in the less objective value than $x_j = u_j$,

$$\frac{1}{2}d_j l_j^2 + \lambda l_j < \frac{1}{2}d_j u_j^2 + \lambda u_j \quad \Rightarrow \quad -d_j(l_j + u_j)/2 < \lambda.$$

- If $x_j = u_j$ results in less objective value than $x_j = l_j$,

$$\frac{1}{2}d_j l_j^2 + \lambda l_j > \frac{1}{2}d_j u_j^2 + \lambda u_j \quad \Rightarrow \quad -d_j(l_j + u_j)/2 > \lambda.$$

- If the objective value is same at $x_j = l_j$ and u_j,

$$\frac{1}{2}d_j l_j^2 + \lambda l_j = \frac{1}{2}d_j u_j^2 + \lambda u_j \quad \Rightarrow \quad \lambda = -d_j(l_j + u_j)/2$$

Thus, $x_j(\lambda)$ for $j \in N$ in the Section 4.2.1 is proved to be correct algebraically.

A.5. Compared solvers

There are a lot of solvers for non-convex quadratic programming. The web page[1] of Gould and Toint lists 28 solvers, and Hans Mittelmann lists more solvers and test problems in his web site[2]. The solvers and options that are used for our experiments are listed below.

[1]http://www.numerical.rl.ac.uk/people/nimg/qp/qp.html
[2]http://plato.asu.edu/sub/nlores.html#QP-problem

A.5.1. Global solver

Recently, Chen and Burer (2012, [23]) implemented their global algorithm for quadratic programming in MATLAB and share it in their website[3]. For fair time comparison, we used an option

```
optChen.verbosity=0;
```

not to display any result during it solves the problem, and added

```
quadopts.Algorithm='active-set';
```

in the code not to display MATLAB's warning message when it use MATLAB as a sub solver. Because it also uses `cplexlp.m` of CPLEX as a sub solver, we added a path to link the solvers in MATLAB. Moreover, we gave the time limit with the option `max_time` in the code. For example, for 30 minutes time limit, the option is made with

```
optChen.max_time=60*30;
```

When the code reaches the time limit, it yields its best optimum solution.

A.5.2. Local solver

As we considered in methodology Section 2.4.1, only three algorithms are available for indefinite case of (P) in literature. Among the methods, we implemented a local solver of Vavasis [132] because the piecewise linear approximation method of Pardalos and Kovoor [101] and $\varepsilon - approximation$ method of [131] get seriously deteriorated as the size of N increases and practically hard to implement as authors never tried it.

Vavasis [132] suggested four versions of algorithms, but he implemented only two versions of `IKP1` ($O(n^3)$ complexity) and `IKP2` ($O(n^2\ log\ n)$ complexity) due to implementation difficulties. Although `IKP2` has lower complexity than `IKP1`, his experiments show that `IKP1` is about 13 times faster than `IKP2` for the size of up to 80. Therefore, we chose `IKP1` to implement as a local solver.

[3]http://dollar.biz.uiowa.edu/~sburer/pmwiki/pmwiki.php%3Fn=Main.QuadprogBB%3Faction=logout.html

A.5.3. Commercial solver

Our algorithms are compared with the commercial solvers of IBM Cplex (version 12.5 and 12.6) and Matlab 2013a. Cplex provides its quadratic programming solver in Matlab with a function name of `cplexqp.m`. Since the version 12.3[4] (June 2011), Cplex has allowed users to obtain a solution that satisfies first order optimality KKT conditions for non-convex problems with an option of

```
optCplex=cplexoptimset('cplex'); optCplex.solutiontarget=2;,
```

and Cplex just launched a global quadratic solver in Dec 2013. The global solver is callable with an option of

```
optCplex=cplexoptimset('cplex'); optCplex.solutiontarget=3;
```

Cplex has various quadratic programming algorithms: Primal Simplex, Dual Simplex, Network Simplex, Barrier, and Concurrent methods, but we let Cplex choose the best algorithm as its default. The time limit option for Cplex can be given by, for 2 hours,

```
optCplex.timelimit=60*60*2;
```

Matlab includes `quadprog.m` as a built-in function for general non-global quadratic programming with three algorithms, but we can use only Active-set method with an option of

```
optsMat=optimset('Algorithm','active-set','Display','off');
```

to solve indefinite case of (P) because its Trust-region-reflective method can solve only bound constrained or linear equality constrained problems, and Interior-point-convex method is only for for convex quadratic programming. The Active-set method we used is also known as a projection method and is based on studies of [47, 46, 45].

[4]One can refer the online manual at http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r3/index.jsp?topic=%2Filog.odms.studio.help
%2FOptimization%2FDocumentation%2FOptimization_Studio%2F_pubskel%2Fps_COS_Eclipse7.html

A.6. Initial expected λ by a sigmoid function

An initial λ for secant method can be expected by the folded logistic regression function

$$b = \frac{1}{1+e^{\lambda}} \in (0,1)$$

adjusting the sigmoid shape in the ranges of $[\lambda_{min}, \lambda_{max}]$ and (b_{min}, b_{max}) where $b_{min} = \sum_{l_j > -\infty} l_j$ and $b_{max} = \sum_{u_j < \infty} u_j$ by

$$\begin{aligned}
b &= \frac{b_{max} - b_{min}}{1 + exp\left[\left(\lambda - \frac{\lambda_{max}+\lambda_{min}}{2}\right)\frac{M}{\lambda_{max}-\lambda_{min}}\right]} + b_{min} \\
\frac{b - b_{min}}{\sum u - b_{min}} &= \frac{1}{1 + exp\left[\left(\lambda - \frac{\lambda_{max}+\lambda_{min}}{2}\right)\frac{M}{\lambda_{max}-\lambda_{min}}\right]} \\
\frac{b_{max} - b_{min}}{b - b_{min}} - 1 &= exp\left[\left(\lambda - \frac{\lambda_{max}+\lambda_{min}}{2}\right)\frac{M}{\lambda_{max}-\lambda_{min}}\right] \\
ln\left(\frac{b_{max} - b_{min}}{b - b_{min}} - 1\right) &= \left(\lambda - \frac{\lambda_{max}+\lambda_{min}}{2}\right)\left(\frac{M}{\lambda_{max}-\lambda_{min}}\right) \\
\lambda &= \left(\frac{\lambda_{max}-\lambda_{min}}{M}\right) ln\left(\frac{b_{max} - b_{min}}{b - b_{min}} - 1\right) + \frac{\lambda_{max}+\lambda_{min}}{2}
\end{aligned}$$

with a large enough M such that

$$M \geq -2ln\left(\frac{\pi}{1-\pi}\right) \quad where\ \pi = \frac{\delta}{b_{max} - b_{min}}$$

Because the sigmoid function never reaches 0 or 1, it cannot be exactly adjusted to pass the points of b_{max} at λ_{min} and b_{min} at λ_{max}. So δ is used to control gap between b_{max} and the point of sigmoid line at λ_{min}, and the same gap is applied for the point at λ_{max}. Thus, with a value of $\delta \in (0, b_{max} - b_{min})$, a large enough M can be obtained by

$$\begin{aligned}
b_{max} - b &= \delta \\
b_{max} - b_{min} &= \left(\frac{b_{max} - b_{min}}{1 + exp\left[\left(\lambda_{min} - \frac{\lambda_{max}+\lambda_{min}}{2}\right)\frac{M}{\lambda_{max}-\lambda_{min}}\right]} \right) + \delta \\
b_{max} - b_{min} &= \left(\frac{b_{max} - b_{min}}{1 + exp(-M/2)} \right) + \delta \\
1 + exp(-M/2) &= 1 + \delta \left(\frac{1 + exp(-M/2)}{b_{max} - b_{min}} \right)
\end{aligned}$$

Let $\pi = \frac{\delta}{b_{max}-b_{min}}$ and note that $\pi < 1$ because $\delta < b_{max} - b_{min}$.

$$\begin{aligned}
1 + exp(-M/2) &= 1 + \pi \left[1 + exp(-M/2)\right] \\
exp(-M/2)\,(1 - \pi) &= \pi \\
exp(-M/2) &= \pi / (1 - \pi) \\
M &= -2ln\left(\frac{\pi}{1 - \pi} \right)
\end{aligned}$$

Note that the last line for M is similar to *-2 Log likelihood ratio* with an odd of π.

A.7. Supplemental experiment results

A.7.1. Chapter 3

Table A.2.: Sensitivity of OBG on (n, δ, ζ, ν)

Solver name	Data size parameter	Solution details	Problem size, n, in millions				
			0.5	0.75	1	1.25	1.5
OBG	$\delta = 10$	cpu sec	27.7	48.5	80.8	157.2	229.0
$(\zeta = 0,$		(cov)	(0.576)	(0.511)	(0.469)	(0.393)	(0.335)
$\nu = 0.5)$	$\delta = 100$	cpu sec	186.8	477.8	524.8	972.1	1247.3
		(cov)	(0.319)	(0.573)	(0.622)	(0.478)	(0.673)
OBG	$\delta = 10$	cpu sec	31.9	93.8	105.2	169.7	207.3
$(\zeta = 0.2,$		(cov)	(0.472)	(0.463)	(0.268)	(0.299)	(0.345)
$\nu = 0.5)$	$\delta = 100$	cpu sec	267.1	573.0	584.5	1113.9	1489.2
		(cov)	(0.419)	(0.399)	(0.604)	(0.243)	(0.332)
OBG	$\delta = 10$	cpu sec	28.6	47.0	107.8	125.7	220.4
$(\zeta = 0,$		(cov)	(0.393)	(0.318)	(0.444)	(0.404)	(0.325)
$\nu = 0.8)$	$\delta = 100$	cpu sec	201.7	427.5	773.3	878.9	1078.7
		(cov)	(0.372)	(0.293)	(0.474)	(0.553)	(0.578)
OBG	$\delta = 10$	cpu sec	22.5	46.9	57.3	99.8	100.4
$(\zeta = 0.2,$		(cov)	(0.414)	(0.333)	(0.445)	(0.353)	(0.355)
$\nu = 0.8)$	$\delta = 100$	cpu sec	135.9	311.5	454.3	686.3	783.8
		(cov)	(0.373)	(0.379)	(0.473)	(0.419)	(0.491)

1. CPX-G is not tested because it spends over 1 hour in the tested sizes.

A.7.2. Chapter 4

A.7.2.1. Experiment results for strictly convex case of (P)

Table A.3.: Range of seconds by size and problem for strictly convex case

	Small coefficient problem							
Size,Problem	1e6	2e6	3e6	4e6	5e6	T1	T2	T3
Bisection	[0.2, 0.2]	[0.4, 0.5]	[0.6, 0.7]	[0.8, 0.9]	[1, 1.1]	[0.2, 1.1]	[0.2, 1]	[0.2, 1]
Sorting	[0.3, 0.4]	[0.7, 0.8]	[1, 1.2]	[1.3, 1.6]	[1.7, 2.1]	[0.3, 2.1]	[0.3, 2]	[0.3, 2]
Median	[0.2, 0.3]	[0.5, 0.6]	[0.7, 0.9]	[1, 1.1]	[1.3, 1.4]	[0.2, 1.4]	[0.2, 1.4]	[0.2, 1.4]
Newton	[0.2, 0.3]	[0.3, 0.5]	[0.5, 0.8]	[0.6, 1.2]	[0.7, 1.4]	[0.2, 1.4]	[0.2, 1.3]	[0.2, 1.3]
Secant	[0.1, 0.1]	[0.2, 0.3]	[0.3, 0.4]	[0.4, 0.5]	[0.4, 0.6]	[0.1, 0.6]	[0.1, 0.6]	[0.1, 0.6]
Interval	[0.2, 0.9]	[0.4, 1.9]	[0.6, 2.8]	[0.9, 3.7]	[1, 4.6]	[0.2, 3.2]	[0.2, 4.6]	[0.3, 3.8]
Sec+Sor	[0.1, 0.2]	[0.2, 0.4]	[0.3, 0.7]	[0.4, 0.9]	[0.6, 1.1]	[0.1, 1.1]	[0.2, 1.1]	[0.2, 1]
Sec+Med	[0.1, 0.2]	[0.2, 0.4]	[0.3, 0.7]	[0.4, 0.9]	[0.6, 1.1]	[0.1, 1.1]	[0.1, 1.1]	[0.2, 1]
Sec+Int	[0.1, 0.2]	[0.2, 0.4]	[0.3, 0.6]	[0.4, 0.8]	[0.5, 1]	[0.1, 1]	[0.1, 1]	[0.1, 1]
New+Med	[0.1, 0.3]	[0.3, 0.6]	[0.5, 1]	[0.6, 1.4]	[0.7, 1.6]	[0.2, 1.6]	[0.2, 1.5]	[0.1, 1.5]
New+Int	[0.2, 0.3]	[0.3, 0.7]	[0.4, 1.1]	[0.6, 1.5]	[0.7, 1.8]	[0.2, 1.8]	[0.2, 1.7]	[0.2, 1.7]
Pegging	[0.3, 0.4]	[0.6, 0.8]	[0.9, 1.2]	[1.2, 1.5]	[1.5, 1.9]	[0.3, 1.9]	[0.3, 1.9]	[0.3, 1.8]
	Large coefficient problem							
Size, Problem	1e6	2e6	3e6	4e6	5e6	T4	T5	T6
Bisection	[0.3, 0.5]	[0.6, 0.9]	[0.9, 1.2]	[1.1, 1.6]	[1.4, 2]	[0.4, 2]	[0.3, 1.5]	[0.3, 1.5]
Sorting	[0.3, 0.5]	[0.7, 0.9]	[1, 1.4]	[1.3, 1.9]	[1.7, 2.4]	[0.4, 2.4]	[0.3, 2]	[0.4, 2.3]
Median	[0.2, 0.4]	[0.5, 0.7]	[0.7, 1]	[1, 1.4]	[1.2, 1.8]	[0.3, 1.8]	[0.2, 1.4]	[0.3, 1.7]
Newton	[0.2, 3.4]	[0.3, 14.8]	[0.4, 4.3]	[0.7, 37.8]	[0.7, 7]	[0.2, 1.7]	[0.2, 1.5]	[0.3, 37.8]
Secant	[0.1, 0.2]	[0.2, 0.4]	[0.3, 0.6]	[0.4, 0.8]	[0.6, 1]	[0.2, 1]	[0.1, 0.7]	[0.2, 0.9]
Interval	[0.2, 0.3]	[0.3, 0.6]	[0.5, 0.9]	[0.6, 1.2]	[0.8, 1.5]	[0.2, 1.3]	[0.2, 1.1]	[0.2, 1.5]
Sec+Sor	[0.1, 0.2]	[0.2, 0.5]	[0.3, 0.7]	[0.4, 1]	[0.6, 1.2]	[0.2, 1.2]	[0.1, 0.9]	[0.2, 1.2]
Sec+Med	[0.1, 0.3]	[0.2, 0.5]	[0.3, 0.7]	[0.4, 0.9]	[0.6, 1.1]	[0.2, 1.1]	[0.1, 0.9]	[0.2, 1.1]
Sec+Int	[0.1, 0.3]	[0.2, 0.6]	[0.3, 0.8]	[0.4, 1.1]	[0.6, 1.3]	[0.2, 1.3]	[0.1, 1.1]	[0.2, 1.3]
New+Med	[0.1, 2.4]	[0.3, 13.3]	[0.4, 4.4]	[0.7, 37.9]	[0.7, 6.9]	[0.2, 2]	[0.1, 1.5]	[0.3, 37.9]
New+Int	[0.1, 2.5]	[0.3, 13.3]	[0.4, 4.5]	[0.7, 38.2]	[0.7, 7.1]	[0.2, 2.1]	[0.1, 1.8]	[0.3, 38.2]
Pegging	[0.2, 0.4]	[0.5, 0.9]	[0.7, 1.4]	[1, 1.9]	[1.2, 2.2]	[0.4, 2.2]	[0.2, 1.6]	[0.4, 2.2]

Table A.4.: Range of iteration by size and problem for strictly convex case

	Small coefficient problem							
Size, Problem	1e6	2e6	3e6	4e6	5e6	T1	T2	T3
Bisection	[27, 29]	[27, 29]	[27, 29]	[27, 29]	[27, 29]	[29, 29]	[27, 28]	[28, 28]
Sorting	[21, 21]	[22, 22]	[23, 23]	[23, 23]	[24, 24]	[21, 24]	[21, 24]	[21, 24]
Median	[21, 21]	[22, 22]	[23, 23]	[23, 23]	[24, 24]	[21, 24]	[21, 24]	[21, 24]
Newton	[4, 8]	[4, 8]	[4, 8]	[4, 9]	[4, 8]	[4, 9]	[4, 8]	[4, 8]
Secant	[8, 14]	[8, 16]	[9, 16]	[9, 16]	[8, 16]	[9, 15]	[8, 16]	[9, 16]
Interval	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]
Sec+Sor	[10, 25]	[12, 26]	[13, 25]	[13, 25]	[13, 26]	[12, 25]	[10, 26]	[13, 21]
Sec+Med	[10, 25]	[12, 26]	[13, 25]	[13, 25]	[13, 26]	[12, 25]	[10, 26]	[13, 21]
Sec+Int	[6, 14]	[6, 15]	[6, 14]	[6, 14]	[6, 15]	[7, 15]	[6, 12]	[8, 12]
New+Med	[4, 12]	[4, 9]	[4, 10]	[4, 10]	[4, 11]	[4, 11]	[4, 12]	[4, 12]
New+Int	[3, 8]	[4, 8]	[4, 8]	[4, 9]	[4, 8]	[4, 9]	[3, 8]	[3, 8]
Pegging	[7, 9]	[7, 10]	[7, 10]	[7, 10]	[7, 10]	[7, 10]	[7, 9]	[7, 9]

	Large coefficient problem							
Size, Problem	1e6	2e6	3e6	4e6	5e6	T4	T5	T6
Bisection	[36, 54]	[36, 54]	[36, 54]	[36, 54]	[36, 54]	[53, 54]	[44, 47]	[36, 36]
Sorting	[21, 21]	[22, 22]	[23, 23]	[23, 23]	[24, 24]	[21, 24]	[21, 24]	[21, 24]
Median	[21, 21]	[22, 22]	[23, 23]	[23, 23]	[24, 24]	[21, 24]	[21, 24]	[21, 24]
Newton	[5, 106]	[5, 267]	[5, 48]	[5, 347]	[5, 45]	[5, 9]	[5, 10]	[5, 347]
Secant	[10, 18]	[10, 18]	[10, 19]	[10, 18]	[10, 19]	[10, 15]	[13, 19]	[10, 16]
Interval	[5, 10]	[5, 10]	[5, 10]	[5, 10]	[5, 10]	[6, 7]	[9, 10]	[5, 6]
Sec+Sor	[8, 18]	[8, 18]	[9, 19]	[9, 18]	[10, 18]	[11, 15]	[13, 19]	[8, 15]
Sec+Med	[8, 18]	[8, 18]	[9, 19]	[9, 18]	[10, 18]	[11, 15]	[13, 19]	[8, 15]
Sec+Int	[8, 18]	[8, 18]	[8, 19]	[8, 18]	[8, 18]	[9, 12]	[13, 19]	[8, 15]
New+Med	[4, 84]	[4, 239]	[5, 48]	[5, 347]	[5, 45]	[5, 9]	[5, 10]	[4, 347]
New+Int	[4, 84]	[4, 239]	[4, 48]	[4, 347]	[4, 45]	[5, 9]	[5, 10]	[4, 347]
Pegging	[5, 9]	[5, 9]	[5, 9]	[5, 9]	[5, 9]	[6, 8]	[5, 7]	[7, 9]

* Iteration for interval test method is only for bracketing phase Algorithm 2.5 of [31]

A.7.2.2. Experiment results for indefinite case of (P)

Table A.5.: Range of seconds by size and problem for indefinite case

	Small coefficient problem							
Size,Problem	1e6	2e6	3e6	4e6	5e6	T1	T2	T3
Bisection	[0.5, 1]	[1, 2]	[1.5, 3]	[2.2, 4]	[2.5, 5.3]	[0.6, 5]	[0.5, 5]	[0.6, 5.3]
Sorting	[0.5, 0.9]	[1, 1.8]	[1.6, 2.7]	[2.1, 3.7]	[2.7, 4.8]	[0.5, 4.6]	[0.5, 4.5]	[0.5, 4.8]
Median	[0.4, 0.8]	[0.9, 1.7]	[1.3, 2.6]	[1.8, 3.4]	[2.3, 4.6]	[0.4, 4.2]	[0.4, 4.2]	[0.4, 4.6]
Newton	[0.3, 1.3]	[0.6, 2.4]	[0.8, 3.6]	[1.2, 4.5]	[1.4, 5.7]	[0.3, 5.4]	[0.3, 5.7]	[0.3, 5.6]
Secant	[0.2, 1.2]	[0.4, 2]	[0.7, 2.9]	[0.8, 3.9]	[1.1, 4.8]	[0.2, 4.3]	[0.2, 4.8]	[0.2, 4.8]
Interval	[0.2, 1.5]	[0.4, 2]	[0.7, 3]	[0.9, 3.7]	[1.1, 4.8]	[0.2, 4.2]	[0.2, 4.8]	[0.2, 3.6]
Sec+Sor	[0.3, 0.9]	[0.5, 1.8]	[0.7, 2.6]	[1, 3.6]	[1.3, 4.5]	[0.3, 3.9]	[0.3, 4.5]	[0.3, 4.2]
Sec+Med	[0.3, 0.9]	[0.5, 1.8]	[0.7, 2.5]	[1, 3.7]	[1.3, 4.5]	[0.3, 3.8]	[0.3, 4.5]	[0.3, 4.2]
Sec+Int	[0.2, 0.5]	[0.5, 1.1]	[0.7, 1.6]	[0.9, 2.2]	[1.2, 3]	[0.2, 2.7]	[0.2, 3]	[0.2, 2.5]
New+Med	[0.3, 1]	[0.6, 2]	[0.8, 3.1]	[1.2, 4.2]	[1.4, 5.1]	[0.3, 4.7]	[0.3, 5.1]	[0.3, 4.9]
New+Int	[0.3, 0.6]	[0.6, 1.2]	[0.8, 1.8]	[1.2, 2.5]	[1.4, 3.1]	[0.3, 2.8]	[0.3, 3.1]	[0.3, 2.8]

	Large coefficient problem							
Size,Problem	1e6	2e6	3e6	4e6	5e6	T4	T5	T6
Bisection	[0.6, 1.7]	[1.1, 3.4]	[1.7, 4.9]	[2.3, 6.5]	[2.9, 8]	[0.7, 8]	[0.7, 7.7]	[0.6, 6.9]
Sorting	[0.5, 0.9]	[1, 1.9]	[1.5, 3]	[2.1, 3.9]	[2.6, 4.9]	[0.6, 4.9]	[0.5, 4.6]	[0.5, 4.9]
Median	[0.4, 0.9]	[0.9, 1.8]	[1.3, 2.8]	[1.8, 3.7]	[2.3, 4.6]	[0.5, 4.6]	[0.4, 4.3]	[0.4, 4.6]
Newton	[0.2, 3.1]	[0.5, 11]	[0.7, 11.2]	[0.9, 10.2]	[1.2, 40.6]	[0.4, 8.9]	[0.2, 8.5]	[0.3, 40.6]
Secant	[0.3, 1.9]	[0.5, 3.9]	[0.7, 6.3]	[1.1, 7.2]	[1.3, 9.9]	[0.3, 7.1]	[0.3, 9.9]	[0.3, 6]
Interval	[0.2, 0.5]	[0.5, 0.9]	[0.7, 1.3]	[1, 1.8]	[1.2, 2.2]	[0.3, 2.2]	[0.2, 2]	[0.3, 2.2]
Sec+Sor	[0.3, 1]	[0.6, 2.1]	[0.8, 3.1]	[1.1, 3.8]	[1.3, 4.4]	[0.3, 3.8]	[0.3, 4.4]	[0.3, 3.3]
Sec+Med	[0.3, 1.1]	[0.6, 2.1]	[0.7, 3.1]	[1.1, 3.8]	[1.3, 4.3]	[0.3, 3.8]	[0.3, 4.3]	[0.3, 3.3]
Sec+Int	[0.3, 1]	[0.5, 2]	[0.7, 2.9]	[1.1, 3.5]	[1.3, 4.1]	[0.3, 3]	[0.3, 4.1]	[0.3, 2.7]
New+Med	[0.2, 1.1]	[0.5, 2.8]	[0.7, 3.2]	[0.9, 5]	[1.2, 5.2]	[0.4, 5.2]	[0.2, 4.8]	[0.3, 5]
New+Int	[0.2, 1.1]	[0.5, 2.6]	[0.7, 3.1]	[0.9, 4.4]	[1.2, 4.9]	[0.4, 4.9]	[0.2, 4.5]	[0.3, 4.4]

Table A.6.: Range of iteration by size and problem for indefinite case

	Small coefficient problem							
Size,Problem	1e6	2e6	3e6	4e6	5e6	T1	T2	T3
Bisection	[28, 52]	[28, 36]	[28, 36]	[28, 54]	[28, 36]	[30, 54]	[28, 52]	[29, 52]
Sorting	[21, 21]	[22, 22]	[22, 23]	[23, 23]	[23, 24]	[21, 24]	[21, 24]	[21, 24]
Median	[21, 21]	[22, 22]	[22, 23]	[23, 23]	[23, 24]	[21, 24]	[21, 24]	[21, 24]
Newton	[6, 47]	[6, 46]	[6, 46]	[7, 43]	[6, 52]	[6, 49]	[6, 52]	[6, 46]
Secant	[8, 41]	[8, 45]	[9, 46]	[7, 42]	[8, 41]	[8, 38]	[7, 41]	[8, 46]
Interval	[3, 4]	[3, 4]	[3, 4]	[3, 4]	[3, 4]	[3, 4]	[3, 4]	[3, 4]
Sec+Sor	[9, 27]	[9, 28]	[8, 27]	[9, 29]	[9, 30]	[10, 25]	[9, 30]	[8, 26]
Sec+Med	[9, 27]	[9, 28]	[8, 27]	[9, 29]	[9, 30]	[10, 25]	[9, 30]	[8, 26]
Sec+Int	[4, 20]	[4, 22]	[6, 20]	[4, 20]	[4, 21]	[4, 17]	[4, 21]	[6, 22]
New+Med	[6, 33]	[6, 35]	[6, 35]	[6, 35]	[6, 48]	[6, 34]	[6, 48]	[6, 34]
New+Int	[4, 21]	[5, 21]	[4, 21]	[5, 20]	[5, 37]	[5, 21]	[4, 37]	[5, 21]
	Large coefficient problem							
Size,Problem	1e6	2e6	3e6	4e6	5e6	T4	T5	T6
Bisection	[36, 79]	[36, 59]	[36, 59]	[36, 81]	[36, 59]	[53, 81]	[42, 55]	[36, 42]
Sorting	[21, 21]	[22, 22]	[22, 23]	[23, 23]	[23, 24]	[21, 24]	[21, 24]	[21, 24]
Median	[21, 21]	[22, 22]	[22, 23]	[23, 23]	[23, 24]	[21, 24]	[21, 24]	[21, 24]
Newton	[3, 93]	[4, 212]	[4, 118]	[4, 87]	[4, 313]	[9, 66]	[3, 122]	[8, 313]
Secant	[8, 70]	[9, 68]	[6, 72]	[8, 66]	[7, 77]	[11, 49]	[6, 77]	[11, 43]
Interval	[4, 10]	[5, 10]	[4, 10]	[5, 10]	[4, 10]	[6, 7]	[4, 10]	[5, 6]
Sec+Sor	[8, 41]	[9, 34]	[7, 36]	[9, 31]	[7, 31]	[11, 21]	[7, 41]	[8, 18]
Sec+Med	[8, 41]	[9, 34]	[7, 36]	[9, 31]	[7, 31]	[11, 21]	[7, 41]	[8, 18]
Sec+Int	[7, 40]	[7, 33]	[6, 35]	[7, 30]	[7, 30]	[9, 17]	[6, 40]	[7, 17]
New+Med	[3, 39]	[4, 55]	[4, 38]	[4, 39]	[4, 37]	[8, 39]	[3, 55]	[6, 39]
New+Int	[3, 36]	[4, 49]	[4, 35]	[4, 35]	[4, 36]	[8, 34]	[3, 49]	[4, 32]

* Iteration for interval test method is only for bracketing phase Algorithm 2.5 of [31]

Table A.7.: Range of the number of breakpoints after bracketing phase

	Small coefficient problem			
% of negative d_j	0	30	60	90
Interval	[411294, 9092367]	[10012, 6287608]	[79, 3000000]	[341888, 4500000]
Sec+Sor, Sec+Med,Sec+Int	[0, 80936]	[0, 47316]	[0, 33136]	[1, 39735]
New+Med, New+Int	[0, 407]	[0, 40774]	[0, 37701]	[41, 69781]
	Large coefficient problem			
% of negative d_j	0	30	60	90
Interval	[27609, 1859865]	[5574, 1558528]	[54, 1266247]	[9285, 2004095]
Sec+Sor, Sec+Med,Sec+Int	[0, 53]	[0, 102]	[0, 110]	[0, 166]
New+Med, New+Int	[0, 18]	[0, 333]	[0, 241]	[0, 264]

Table A.8.: Average seconds by % of negative d_j

	Small coefficient problem					Large coefficient problem			
% of negative d_j	0	30	60	90	% of negative d_j	0	30	60	90
Sec+Int	0.53	0.85	1.05	1.13	Interval	0.69	1.02	1.13	1.16
New+Int	0.80	1.06	1.16	1.38	Sec+Int	0.64	1.12	1.39	1.60
Secant	0.32	0.89	1.50	2.10	Sec+Med	0.59	1.15	1.54	1.88
Sec+Med	0.57	1.12	1.58	1.99	Sec+Sor	0.59	1.15	1.55	1.88
Sec+Sor	0.57	1.12	1.58	2.00	New+Int	1.05	1.68	1.45	1.70
Interval	1.53	1.30	1.27	1.78	New+Med	0.98	1.78	1.64	2.04
New+Med	0.72	1.16	1.65	2.52	Median	0.90	1.58	2.03	2.38
Newton	0.63	1.17	1.76	2.51	Secant	0.47	1.33	2.15	3.22
Median	0.79	1.42	1.89	2.30	Sorting	1.20	1.83	2.25	2.55
Bisection	0.62	1.68	2.20	2.74	Newton	0.99	2.53	2.65	3.45
Sorting	1.08	1.67	2.09	2.47	Bisection	0.99	2.25	3.10	3.85

* Lists are sorted in total time for each size of coefficients.

B. Implementation in Matlab

In this dissertation, all algorithm implementations and experiments are done in MATLAB, but there are issues in implementation due to MATLAB's features such as Just-in-Time compilation, copy-on-write type editing, and fast full vector computation. Considering the issues, we present why we do not include fixing Algorithm 4.5 in our code for `CBS` and what sorting and median algorithms, which are main sub procedure for sorting, medians search, and interval test methods, are employed in our implementation.

B.1. Implementation in Matlab

The performance of sorting, median search, and interval test methods in Chapter 4 are mainly dependent on sorting or median algorithms, which dominate complexity of the methods. So we need to choose an efficient sorting and median algorithms to implement methods.

Sorting algorithm

According to the book of Press and Teukolsky (2007, Numerical recipe [108]), "For large N (say > 1000), `QUICKSORT` is faster, on most machines, by a factor of 1.5 or 2; it requires a bit of extra memory, however, and is a moderately complicated program" although its worst case complexity is $O(n^2)$ and average case performance is $O(n\ log\ n)$. For this reason, most literature that considered sorting method used `QUICKSORT` of Hoare [51] for their implementation and experiments. MATLAB has a built-in sorting function `sort.m`, which implements `QUICKSORT` (See the web site manual [1]). Thus, we use the MATLAB's built-in `sort.m` for all sorting procedure in our implementation.

[1]http://www.mathworks.com/support/solutions/en/data/1-15K1B/

Median search algorithm

For the median algorithm, it has been proved that `SELECT` of Floyd and Rivest [43] performs best in practice by Kiwiel [66], and the book of Press and Teukolsky (2007, Numerical recipe [108]) also mentions about `SELECT` as "the fastest general method for selection, allowing rearrangement, is partitioning, exactly as was done in the `QUICKSORT` algorithm." Thus, `SELECT` may be the best choice for our implementation. However, MATLAB's built-in median function `median.m` does not use `SELECT`. Instead, it first sorts entire elements in an array using its `sort.m` function and picks the median[2] so median search method cannot have superiority over sorting method if `median.m` is used.

We can implement the algorithm of `SELECT` in MATLAB, but it may not be efficient because median search method uses it multiple times in every iteration, and generally such a sub function performs significantly better if it is coded in `C/C++` or FORTRAN and complied for MATLAB with `MEX` complier. Fortunately, Peter Li shares his `MEX` codes through a web site of Matlab central[3], and the code allows to compile the `C++` standard template library `nth_element` into `MEX` MATLAB function. The template code `nth_element` guarantees the $O(n)$ expected time as described in the technical report [56] of International Standardization Organization (ISO).

In the shared files of Peter Li, there are two types of median functions: `fast_median.m` for copy-on-write type and `fast_median_ip.m` for in-place type. The in-place type is generally faster and consume less memory than the copy-on-write type because the in-place type directly swaps memory address to rearrange elements on the left half for ones that are less than median and on the right for the rest of ones, while the copy-on-write type first copies the input data in a separate memory and finds the median in the same way. In our experiments, for an array with the size of $n = 10^6$, `fast_median_ip.m` is about 5 times faster than MATLAB' `median.m` and about 1.3 times faster than `fast_median.m` on average.

Although MATLAB persists copy-on-write type for its built-in functions, the in-place type coincides with the purpose in median search method because we can exclude half of break-

[2]One can type "`edit median`" on MATLAB's command window to see the MATLAB's built-in `median.m` code.
[3]http://www.mathworks.com/matlabcentral/fileexchange/29453-nthelement

points simply selecting first half or last half of the breakpoint array after finding median by `fast_median_ip.m`. Therefore, we use `fast_median_ip.m` to implement median search method.

Time measurement; `tic/toc`

Since a main performance measurement is the elapsed time of solvers, we carefully chose a method to measure time. MATLAB has two time measurement functions, `cputime` and `tic/toc`. `cputime` records CPU time that MATLAB actually spends for its operations, while `tic/toc` functions just like a stop watch timer. So `tic/toc` may be sensitively influenced by the computer's environment. However, according to the online manual[4] of MATLAB, `tic/toc` is recommended especially for WINDOWS operations that we used for experiments because `cputime` can fail to measure proper time if it is used in hyperthreading running WINDOWS system. Therefore, all time records in our experiments are measured by `tic/toc`.

B.2. Efficient coding in Matlab

In our suggested hybrid methods in Chapter 4, we include fixing algorithm 4.5 because it efficiently reduces the size of problem fixing variables at its one of extreme points every iteration; however, it does not improve the performance if the methods are implemented in MATLAB because it is a dynamic programming language. While the third generation programming language (3GL) such as `C/C++` or FORTRAN compiles before it executes, MATLAB a fourth generation programming languages (4GL) compiles its code for each line dynamically so we call it Just-in-Time (JIT) compilation. It is very convenient when programmers make prototype implementations because it interactively gives feedback for each line and is easy to modify array size and data type during execution.

As a JIT programming language, MATLAB supposedly does not have pointers to access a memory directly with a memory address, and instead it copies the data allocating new memory temporarily when a part of an array is used and then releases it from memory. The

[4]http://www.mathworks.com/help/matlab/matlab_prog/analyzing-your-programs-performance.html

time for this operation is very considerable especially when the size is big and even results worse if it needs to use virtual memory and page swaps.

However, the pure methods and proposed hybrid methods need to use pointers very frequently to implement fixing algorithm 4.5 and get $\mathbf{x}(\lambda)$ every iteration because each variables can be calculated by its separate coefficients. We found three indexing options which can role as pointers in Matlab;

- Numerical index
- Logical index
- Handle object trick.

The first easy idea is numerical integer index set. Matlab allows to define four kinds of unsigned integers[5] by `uint8`, `uint16`, `unit32`, `uint64`. Each number after `int` represents the size of bits for each element and maximum allowable integer number; for example, we can hold integer numbers from 0 to $2^8 - 1$ by `uint8` with 1 byte (= 8bits) for each element. So it may be enough to use `unit32` for experiments because it can hold an index number up to about $4.29e9$. The second option is logical index so called boolean array or true/false array. Each element in a logical array is a binary number of either 0 or 1 consuming 1 bit. In the respect to the efficient memory usage, it is not obvious to choose one of two options. Suppose we solve (P) for n variables. Then, memory has to be allocated for an index set with $4n$ bytes if `unit32` integer index is used and with n bytes if logical index is used. Initially the numerical index consumes 4 times more memory than the logical index, but it gradually reduces its memory usage and consumes less memory when the number of unfixed variables are less than $n/4$, while logical index always hold n bytes. The third option is a trick utilizing `classdef` and `handle`. As in the online Matlab manual[6], we can define a new class that allows to access memory directly as pointer does.

However, we have to consider another feature of Matlab before we choose one of three indexing options. It is generally known that Matlab is highly specialized for vector and

[5]http://www.mathworks.com/help/matlab/matlab_prog/integers.html

[6]http://www.mathworks.com/help/matlab/matlab_oop/creating-object-arrays.html#brd4nrh and for a simple example see http://www.matlabtips.com/how-to-point-at-in-matlab/

matrix computation and the speed is dramatically faster than loops such as `for-loop` and `while-loop`. To overcome the drawback, MATLAB has developed and improved `JIT accelerator` since version 6.5 (2002). It sometimes makes `for-loop` faster than vector computation, but it works only when some strict conditions are satisfied (see [81] for conditions) and any official manual has not been published due to its imperfection. So we have observed that `for-loop` with all three indexing options is significantly slower than full vector computation during we implement methods in MATLAB.

Alternatively we may use vector computation with an index set as MATLAB generally recommends in its manual. For example, if we want to get $s = \sum_{j \in I} a_j b_j$ with a subset $I \subseteq \{1, 2..., n\}$, we can code it by

```
s=a(I)'*b(I)
```

in MATLAB instead of `for-loop`. However, full vector computation

```
s=a'*b
```

is still much faster than (B.2).

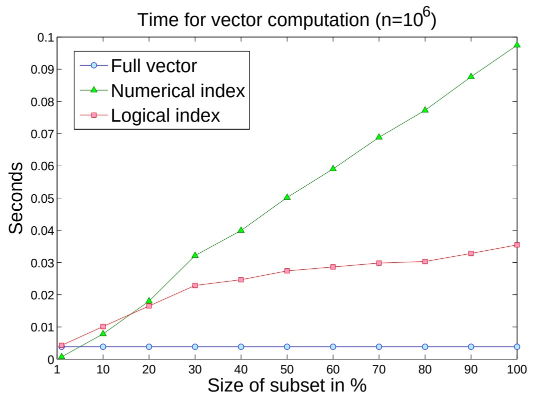


Figure B.1.: Computation time for $s = \sum_{j \in I} a_j b_j$

Figure B.1 shows the average computation time of (B.2) with numerical indexing and logical indexing and (B.2) on various random subset size of 1% to 100% in $n = 1e6$. The full vector computation is extremely faster than other indexing methods when the size of subset is over approximately 5% for numerical indexing and is over approximately 1% for logical indexing. It is because of JIT compilation which abandons pointers. When MATLAB calls a part of array with an index set, it first check if the index is within the array and then allocate temporary memory to copy the indexed elements and release the memory after computation is finished. The time for the operations are not negligible and actually significant when the array size is large.

Logical indexing is initially slower than numerical indexing, but it becomes much faster when the size of subset is over approximately 15%. In addition to the speed advantage, logical indexing is proper for our implementation because multiple index sets are used in our codes and many index numbers are frequently in and out every iteration. If numerical indexing is used, the size of memory for index sets fluctuates; thus, it harms the performance of implementation while it dynamically allocate and release memory, and it may results fragmented memory that also deteriorate the performance regardless of methods that the codes implement. Therefore, we do not use fixing algorithm 4.5 in our implementation and pursue the full vector computation and logical indexing coding style unless numerical indexing is apparently better.

Vita

Jaehwan Jeong was born in Seoul, Republic of Korea (South Korea) on November 11, 1981. He received a Bachelor of Science degree in Business Administration from Dongguk University at Seoul in August 2006. During his undergraduate period, he served Republic of Korea Army in February 2002 - March 2004. In August 2007, he entered the master program in the department of Statistics, Operations, and Management Science in the University of Tennessee, Knoxville, USA. After receiving the Master of Science degree in August 2009 (major in Management Science and minor in Statistics), he continued his study in the same department and received the Doctoral degree of Management Science in May 2014.