mathematics

MDPI

Article

An Active-Set Algorithm for Convex Quadratic Programming Subject to Box Constraints with Applications in Non-Linear Optimization and Machine Learning

Konstantinos Vogklis [1,*,†] and Isaac E. Lagaris [2,†]

[1] Department of Tourism, Ionian University, 49100 Kerkira, Greece
[2] Department of Computer Science and Engineering, University of Ioannina, 45110 Ioannina, Greece; lagaris@uoi.gr
* Correspondence: voglis@ionio.gr
† These authors contributed equally to this work.

Abstract: A quadratic programming problem with positive definite Hessian subject to box constraints is solved, using an active-set approach. Convex quadratic programming (QP) problems with box constraints appear quite frequently in various real-world applications. The proposed method employs an active-set strategy with Lagrange multipliers, demonstrating rapid convergence. The algorithm, at each iteration, modifies both the minimization parameters in the primal space and the Lagrange multipliers in the dual space. The algorithm is particularly well suited for machine learning, scientific computing, and engineering applications that require solving box constraint QP subproblems efficiently. Key use cases include Support Vector Machines (SVMs), reinforcement learning, portfolio optimization, and trust-region methods in non-linear programming. Extensive numerical experiments demonstrate the method's superior performance in handling large-scale problems, making it an ideal choice for contemporary optimization tasks. To encourage and facilitate its adoption, the implementation is available in multiple programming languages, ensuring easy integration into existing optimization frameworks.

Keywords: convex quadratic programming; machine learning; optimization; active set; Lagrange multipliers; practical applications

MSC: 65K10

Academic Editor: Yu Liang

Received: 18 March 2025
Revised: 16 April 2025
Accepted: 28 April 2025
Published: 29 April 2025

Citation: Vogklis, K.; Lagaris, I.E. An Active-Set Algorithm for Convex Quadratic Programming Subject to Box Constraints with Applications in Non-Linear Optimization and Machine Learning. *Mathematics* **2025**, *13*, 1467. https://doi.org/10.3390/math13091467

1. Introduction

Convex quadratic programming (QP) refers to optimization problems in which a quadratic objective function is minimized subject to linear constraints. In the convex case, the quadratic cost matrix is positive semidefinite, ensuring a unimodal (bowl-shaped) objective so that any local minimum is global. The convex quadratic programming problem with simple bounds is stated as:

$$\min_x \ \frac{1}{2}x^T Bx + x^T d, \text{ subject to: } a_i \leq x_i \leq b_i, \forall i \in I = \{1, 2, \cdots, N\} \tag{1}$$

where $x, d \in R^N$ and B are a symmetric, positive definite $N \times N$ matrix. In this paper, we propose a novel active-set strategy for solving problems of the form in Equation (1), that in each step solves a linear system and updates the active set and the Lagrange multipliers accordingly.

This type of problem minimizing a convex quadratic function subject to bound constraints appears quite frequently in scientific applications as well as in parts of generic optimization algorithms. Many non-linear optimization techniques are based on solving quadratic model subproblems [1–4].

While much of the foundational work on convex quadratic programming with box constraints originates from earlier research, there has also been a growing body of recent literature proposing modifications and new approaches targeting higher efficiency [5–8].

Portfolio optimization problems often use QP to allocate assets efficiently [9–12]. In Markowitz's classic mean-variance portfolio model, the objective is quadratic (portfolio risk) and constraints ensure that the allocation weights sum to a budget [10]. If short-selling is disallowed (no negative weights), this imposes bound constraints $x_i \geq 0$, rendering the model convex QP. Such QP formulations are central in financial risk management and asset allocation.

Support Vector Machines (SVMs) for classification and regression are trained by solving a convex QP that maximizes the margin between classes while penalizing errors. The SVM optimization typically includes linear constraints and bound constraints on the variables of the dual formulation (e.g., $0 \leq \alpha_i \leq C$ in the dual formulation) [13–16]. Specialized solvers like the sequential minimal optimization (SMO) algorithm exploit this structure by breaking the problem into small QP subproblems. Similarly, in data-fitting applications, *non-negative least squares* (NNLS) problems (which minimize a sum of squared errors with $x_i \geq 0$) are convex QPs with simple bounds. NNLS [17,18] and its generalization to bounded-variable least squares (constraining $\alpha_i \leq x_i \leq \beta_i$) are widely used in machine learning and signal processing.

Many engineering optimization tasks are naturally cast as QPs with bounded controls or resources. For instance, in model predictive control (MPC) [19,20]—a technique for controlling processes and robots—the controller solves a QP at each time step to minimize a quadratic performance index, subject to constraints like actuator limits and state bounds. These actuator limits are simple bound constraints on the control variables. Likewise, energy management systems, network flow optimization, and other resource allocation problems often involve quadratic cost functions (e.g., minimizing power loss or deviation from a target) with variables constrained by minimum/maximum operating levels.

The optimization of radiation intensity in oncology treatment [21] is a critical aspect of radiotherapy planning, aiming to maximize the dose delivered to tumor regions while minimizing exposure to surrounding healthy tissues. These methods are commonly formulated as quadratic optimization problems, where the objective function represents the trade-off between dose conformity and tissue sparing. The quadratic nature arises from the squared deviation of the delivered dose from the prescribed dose, ensuring a smooth and controlled distribution of radiation. Constraints are incorporated to enforce clinical requirements such as maximum allowable dose limits for organs at risk and minimum dose thresholds for tumor coverage. Various numerical techniques, including gradient-based algorithms and interior-point methods, are employed to efficiently solve these optimization problems, ensuring precise and effective treatment planning.

For the problem in Equation ((1)) two major strategies exist in the literature, both of which require feasible steps to be taken. The first one is the active-set strategy [2,3], which generates iterates on a face of the feasible box, never violating the primary constraints on the variables. Active-set algorithms work by iteratively guessing which constraints are "active" and which are not. They temporarily treat the active constraints as equalities, solve the resulting reduced QP, then check optimality conditions (Karush–Kuhn–Tucker conditions) to adjust the active set.

The basic disadvantage of this approach, especially in the large-scale case, is that constraints are added or removed one at a time, thus requiring a number of iterations proportional to the problem size. To overcome this, gradient projection methods [22,23] were proposed. In that framework, the active set algorithm is allowed to add or remove many constraints per iteration.

The second strategy treats the inequality constraints using interior-point techniques. In brief, an interior-point algorithm consists of a series of parametrized barrier functions which are minimized using Newton's method. The major computational cost is due to the solution of a linear system, which provides a feasible search direction. Modern primal–dual interior-point algorithms are known for their polynomial-time complexity and strong practical performance on large-scale problems. In contrast to active-set methods (which pivot along constraint boundaries), interior-point methods move through the interior and typically require fewer iterations (albeit with more computation per iteration).

In this paper, we investigate a series of convex quadratic test problems. We recognize that bound constraints are a very special case of linear inequalities, which may in general have the form $Ax \geq b$, with A being an mxn matrix and b a vector $\in R^m$. Our investigation is also motivated by the fact that in the convex case, every problem subject to inequality constraints can be transformed to a bound-constrained one, using duality [24,25]:

$$\begin{array}{lcl} \min\limits_{x \in R^n} \dfrac{1}{2} x^T B x + x^T d & \xrightarrow{\tilde{B}=AB^{-1}A^T,\ \tilde{d}=AB^{-1}d+b} & \min\limits_{y \in R^m} \dfrac{1}{2} y^T \tilde{B} y + y^T(-\tilde{d}) \\ \text{subject to: } Ax \geq b & \xleftarrow{x^*=B^{-1}(A^T y^* - d)} & \text{subject to: } 0 \leq y \leq Inf \end{array} \tag{2}$$

It is important to note, however, that the dual transformation from a general inequality-constrained convex quadratic program to a bound-constrained dual problem does not always result in a positive definite matrix $\tilde{B} = AB^{-1}A^\top$. The positive definiteness of $\tilde{B}$ is guaranteed only when the matrix A has full row rank and the original Hessian matrix B is strictly positive definite. If A is rank-deficient or if B is only positive semi-definite, the resulting $\tilde{B}$ may become singular or indefinite, which can negatively affect the stability and solvability of the dual problem.

Our proposed approach to solving the problem in Equation (1) is an active-set algorithm that, unlike traditional methods, does not require strictly feasible or descent directions at each iteration. While an initial version of the algorithm was briefly introduced in [26], this paper presents extended results and a detailed comparison. We provide a step-by-step description of the algorithm, including the Lagrange multiplier updates, dual feasibility checks, and implementation insights. Furthermore, we demonstrate the algorithm's broad applicability and its consistently rapid convergence through extensive experiments on a diverse set of benchmark problems, including synthetic, structured, and real-world optimization tasks. Comparative results and a ranking-based evaluation confirm the robustness and efficiency of our method across a host of problems of varying dimensions and conditioning.

The paper is organized as follows. The proposed algorithm is described in detail in Section 2. In Section 3, we briefly present four quadratic programming codes that were used against our method on five different problem types, which are described in Section 4. Finally, in Section 5.6, a new trust-region-like method is proposed which takes full advantage of our quadratic programming algorithm.

2. Solving the Quadratic Problem

For the problem in Equation (1), we introduce Lagrange multipliers in order construct the associated Lagrangian:

$$L(x, \lambda, \mu) = \frac{1}{2} x^T B x + x^T d - \lambda^T (x - a) - \mu^T (b - x) \tag{3}$$

The necessary optimality conditions (KKT conditions) at the minimum x^*, λ^*, μ^* require that:

$$Bx^* + d - \lambda^* + \mu^* = 0 \ \{\textit{first order stationarity condition }\} \tag{4a}$$

$$x_i^* \in [a_i, b_i], \ \forall i \in I \ \{\textit{primal feasibility }\} \tag{4b}$$

$$\lambda_i^* \geq 0, \ \forall i \in I \ \{\textit{dual feasibility (lower bound) }\} \tag{4c}$$

$$\mu_i^* \geq 0, \ \forall i \in I \ \{\textit{dual feasibility (upper bound) }\} \tag{4d}$$

$$\lambda_i^* (x_i^* - a_i) = 0, \ \forall i \in I \{\textit{complementarity slackness (lower bound)}\} \tag{4e}$$

$$\mu_i^* (b_i - x_i^*) = 0, \ \forall i \in I \ \{\textit{complementarity slackness (upper bound) }\} \tag{4f}$$

A solution to all the equations of above system (4) can be obtained through an active-set strategy sketched in the following steps:

1. At the initial iteration, we set the Lagrange multipliers μ and λ to zero and compute the Newton point $x = -B^{-1}d$.
   If $x^{(0)}$ is feasible, it is accepted as the optimal solution.
2. At each iteration k, we define three disjoint index sets:
   (a) L: indices where the lower bound is active or violated (Equation (4b));
   U: indices where the upper bound is active or violated (Equation (4b));
   S: indices where x is strictly feasible and no bound constraints are active (Equation (4b)).
   (b) For each $i \in L$, the corresponding variable x_i is set to the lower bound, satisfying primal feasibility (Equation (4b)), and μ_i is set to zero, satisfying the complementarity condition (Equation (4e)).
   (c) For each $i \in U$, the value x_i is set to the upper bound, satisfying primal feasibility (Equation (4b)), and λ_i is set to zero, again satisfying the complementarity condition (Equation (4f)).
   (d) For all $i \in S$, where x_i is strictly within bounds, both multipliers μ_i and λ_i are set to zero, satisfying complementarity conditions (Equations (4e) and (4f)).
   (e) The rest of the N unknowns—namely the λ_i for $i \in L$, the μ_i for $i \in U$, and the x_i for $i \in S$—are computed by solving the stationarity condition after some rearrangement:
   $$Bx + d = \lambda - \mu.$$

The BoxCQP (abbreviation for box-constrained quadratic programming) algorithm is formally presented below:

The solution of the linear system in Step 3 of Algorithm 1 needs further consideration. Let us rewrite the system in a component-wise fashion.

$$\sum_{j \in I} B_{ij} x_j^{(k+1)} + d_i = \lambda_i^{(k+1)} - \mu_i^{(k+1)}, \ \forall i \in I \tag{5}$$

Since $\forall i \in S^{(k)}$, we have that $\lambda_i^{(k+1)} = \mu_i^{(k+1)} = 0$; hence, we can calculate $x_i^{(k+1)}$, $\forall i \in S^{(k)}$ by splitting the sum in Equation (5) and taking into account Step 2 of the algorithm, i.e.,:

$$\sum_{j \in S^{(k)}} B_{ij} x_j^{(k+1)} = -\sum_{j \in L^{(k)}} B_{ij} a_j - \sum_{j \in U^{(k)}} B_{ij} b_j - d_i, \ \forall i \in S^{(k)} \tag{6}$$

The submatrix B_{ij}, with $i, j \in S^{(k)}$ is positive definite as can be readily verified, given that the full matrix B is. The calculation of $\lambda_i^{(k+1)}$, $\forall i \in L^{(k)}$ and of $\mu_i^{(k+1)}$, $\forall i \in U^{(k)}$ is straightforward and is given by:

$$\lambda_i^{(k+1)} = \sum_{j \in I} B_{ij} x_j^{(k+1)} + d_i, \ \forall i \in L^{(k)} \tag{7}$$

$$\mu_i^{(k+1)} = -\sum_{j \in I} B_{ij} x_j^{(k+1)} - d_i, \ \forall i \in U^{(k)} \tag{8}$$

Algorithm 1 BoxCQP

Initially set: $k = 0$, $\lambda^{(0)} = \mu^{(0)} = 0$ and $x^{(0)} = -B^{-1}d$.
If $x^{(0)}$ is feasible, **Stop**, the solution is: $x^* = x^{(0)}$.
At iteration k, the quantities $x^{(k)}, \lambda^{(k)}, \mu^{(k)}$ are available.

1. Define the sets:

$$\begin{aligned} L^{(k)} &= \{i : x_i^{(k)} < a_i, \text{ or } x_i^{(k)} = a_i \text{ and } \lambda_i^{(k)} \geq 0\} \\ U^{(k)} &= \{i : x_i^{(k)} > b_i, \text{ or } x_i^{(k)} = b_i \text{ and } \mu_i^{(k)} \geq 0\} \\ S^{(k)} &= \{i : a_i < x_i^{(k)} < b_i, \text{ or } x_i^{(k)} = a_i \text{ and } \lambda_i^{(k)} < 0, \\ &\text{or} \quad x_i^{(k)} = b_i \text{ and } \mu_i^{(k)} < 0\} \end{aligned}$$

   where $L^{(k)} \cup U^{(k)} \cup S^{(k)} = I$

2. Set:

$$\begin{aligned} x_i^{(k+1)} &= a_i, \ \mu_i^{(k+1)} = 0, \ \forall i \in L^{(k)} \\ x_i^{(k+1)} &= b_i, \ \lambda_i^{(k+1)} = 0, \ \forall i \in U^{(k)} \\ \lambda_i^{(k+1)} &= 0, \ \mu_i^{(k+1)} = 0, \ \forall i \in S^{(k)} \end{aligned}$$

3. Solve:

$$Bx^{(k+1)} + d = \lambda^{(k+1)} - \mu^{(k+1)}$$

   for the N unknowns: $x_i^{(k+1)}, \forall i \in S^{(k)}$, $\mu_i^{(k+1)}, \forall i \in U^{(k)}$, $\lambda_i^{(k+1)}, \forall i \in L^{(k)}$

4. Check if the new point is a solution and decide to either stop or iterate.

   If $x_i^{(k+1)} \in [a_i, b_i] \ \forall i \in S^{(k)}$ **and** $\mu_i^{(k+1)} \geq 0$, $\forall i \in U^{(k)}$ **and** $\lambda_i^{(k+1)} \geq 0$, $\forall i \in L^{(k)}$ **Then**
   Stop; the solution is: $x^* = x^{(k+1)}$.
   Else
   set $k \leftarrow k + 1$ **and iterate from Step 1.**
   Endif

The convergence analysis along the lines of Kunisch and Rendl [27] is applicable for our method as well. Hungerländer and Rendl [5] have showed that when the Hessian B is positive definite, then there exists a solution, and have developed a procedure leading to a convergence proof. One may also apply their scheme to prove the convergence of the

presented algorithm. However, it is lengthy and complicated, and therefore we preferred to present extended numerical evidence instead. We numerically tested cases with thousands of variables and a wide range for the condition number of B from 1 to 10^{20}. When B becomes nearly singular, then cycling occurs as expected. (Note that in such a case, the linear system $Bx = -d$ is ill-conditioned). At this point, ad hoc corrective measures may be taken.

The main computational task of the algorithm above is the solution of the linear system in Step 3. We have implemented three variants that differ in the way the linear system is solved. In Variant 1, at every iteration, we use a Cholesky LDL^T decomposition. In Variant 2, we employ a conjugate gradient iterative method [28,29] throughout. In Variant 3, at the first iteration, where we need to solve the full system, we use a few iterations of the conjugate gradient scheme and subsequently LDL^T decomposition.

Table 1 provides a summary of the KKT conditions that are assured to be met at each constructive iteration, classified according to the respective index sets (L, S, U). Most of the six KKT conditions are persistently adhered to throughout the procedure. Nonetheless, certain indices might momentarily breach conditions such as primal feasibility (4b), lower-bound dual feasibility (4c), or upper-bound dual feasibility (4d). Crucially, the number of discrepancies involving either Lagrange multipliers or primal variables remains below the problem dimension N. This table outlines the dynamic modification of the primal and dual variables and demonstrates their changing relationship with the KKT conditions as the algorithm progresses.

Table 1. BoxCQP consecutive iterations and KKT.

Iteration k	Iteration $k+1$	Guaranteed Satisfied KKT $k+1$	Not Satisfied KKT $k+1$
$i \in L$	$x_i \leftarrow a_i$, $\mu_i \leftarrow 0$, λ_i from Equation (4a)	(4a), (4b), (4d), (4e), (4f)	(4c)
$i \in U$	$x_i \leftarrow b_i$, $\lambda_i \leftarrow 0$, μ_i from Equation (4a)	(4a), (4b), (4c), (4e), (4f)	(4d)
$i \in S$	$\lambda_i \leftarrow 0$, $\mu_i \leftarrow 0$, x_i from Equation (4a)	(4a), (4c), (4d), (4e), (4f)	(4b)

Experimental Convergence Analysis: Controlled Indefiniteness

To assess the robustness of the BoxCQP algorithm beyond its scope, we designed a controlled experiment that systematically introduces indefiniteness into the quadratic term of the objective function. We argue that BoxCQP algorithm converges for strictly positive definite matrices B. Although theoretical convergence is possible following the proof found in [5], it is also imperative to investigate the behavior of the algorithm in a practical manner. To examine its behavior under near-indefinite and indefinite scenarios, we generated perturbed matrices that violate this assumption in a controlled way.

The experimental procedure described next was repeated for 100 random instances for every dimension setting:

Step 1: Matrix and Vector Generation:
For a given problem dimension $n \in \{10, 100, 1000, 5000\}$, we generated a random positive definite matrix $B \in \mathbb{R}^{n \times n}$, as well as random vectors $d \in \mathbb{R}^n$, $x_l \in \mathbb{R}^n$, and $x_u \in \mathbb{R}^n$, defining the box-constrained quadratic programming problem:

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} x^\top B x + d^\top x \quad \text{s.t.} \quad x_l \leq x \leq x_u.$$

Step 2: Controlled Perturbation:
We applied a Cholesky decomposition $B = LL^\top$. Then, approximately 20% of the diagonal entries of L were selectively modified by replacing them with values drawn from the set:

$$0,\ \pm 0.1,\ \pm 0.01,\ \pm 0.001,\ \pm 0.0001,\ \pm 0.00001,\ \pm 10^{-8},\ \pm 10^{-12},\ \pm 10^{-16}$$

This range includes small negative, zero, and small positive values, effectively creating a smooth spectrum from definiteness to indefiniteness. The modified lower-triangular matrix $\tilde{L}$ was then used to reconstruct $\tilde{B} = \tilde{L}\tilde{L}^\top$, which served as the input matrix for BoxCQP.

Step 3: Solver Execution:
Each problem instance $(\tilde{B}, d, x_l, x_u)$ was solved using the BoxCQP algorithm with a maximum limit of 100 iterations. For each configuration, we recorded the number of iterations required to converge, or marked the instance as failed if convergence was not reached within the iteration limit.

Step 4: Evaluation Metrics:
For each perturbation level and each dimension, we measure:

- The mean number of iterations required to reach convergence;
- The failure count, i.e., the number of cases out of 100 in which the algorithm did not converge;
- The average condition number of the resulting matrix $\tilde{B}$, measured as the ratio of its largest to smallest eigenvalue.

This procedure allowed us to systematically investigate how BoxCQP performs as the definiteness of the matrix degrades, and to associate failure patterns and convergence delays with condition number growth and specific types of matrix perturbations. The results are shown in Table 2 and graphically in Figure 1.

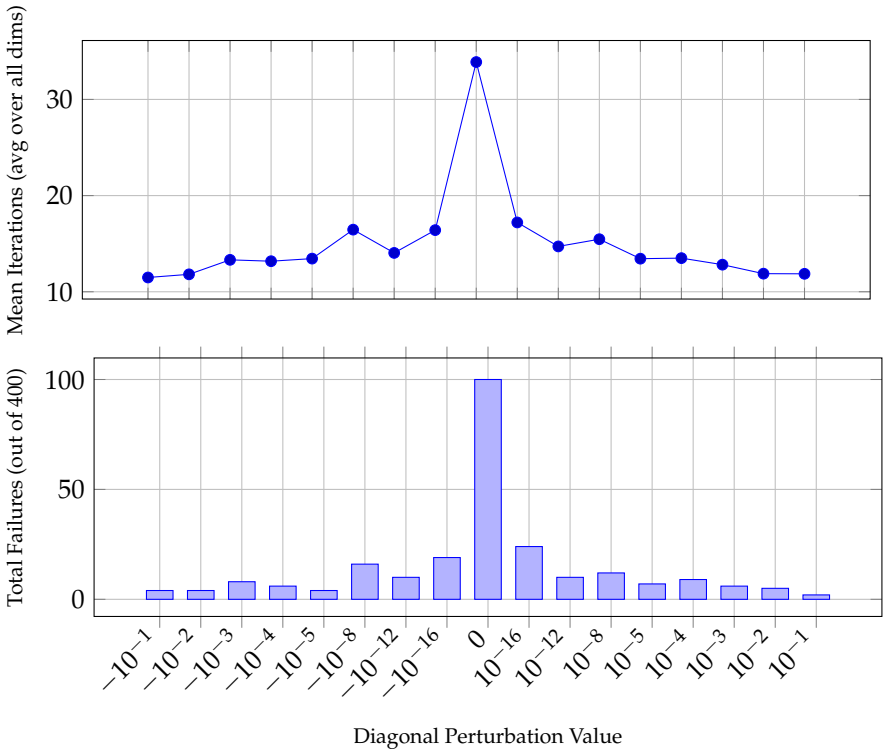


Figure 1. BoxCQP convergence vs. perturbation level.

Table 2. BoxCQP performance under controlled indefiniteness: mean iterations, failure counts, and condition numbers (100 runs).

Min Diag	$n = 10$			$n = 100$			$n = 1000$			$n = 5000$		
	Mean It.	**Fail (%)**	**Cond**	**Mean It.**	**Fail (%)**	**Cond**	**Mean It.**	**Fail (%)**	**Cond**	**Mean It.**	**Fail (%)**	**Cond**
-1×10^{-1}	8.88	3	3.86×10^{5}	10.48	3	2.53×10^{16}	13.00	0	8.60×10^{18}	13.58	0	1.61×10^{20}
-1×10^{-2}	9.15	3	3.04×10^{8}	10.38	3	7.89×10^{18}	13.06	0	1.84×10^{19}	14.66	1	7.81×10^{19}
-1×10^{-3}	11.79	6	1.26×10^{9}	12.12	1	1.56×10^{18}	12.95	0	9.96×10^{20}	16.42	1	4.70×10^{19}
-1×10^{-4}	10.02	4	1.11×10^{11}	11.40	1	1.75×10^{21}	14.44	0	1.72×10^{19}	16.85	1	7.63×10^{22}
-1×10^{-5}	9.47	3	2.45×10^{13}	11.23	0	2.92×10^{20}	15.93	1	4.73×10^{19}	17.16	1	5.95×10^{19}
-1×10^{-8}	17.20	11	3.54×10^{17}	14.13	1	3.83×10^{25}	14.65	0	3.89×10^{25}	19.86	3	8.57×10^{27}
-1×10^{-12}	11.48	5	1.76×10^{17}	12.95	1	4.32×10^{28}	13.54	0	6.82×10^{18}	18.19	4	1.23×10^{20}
-1×10^{-16}	17.12	11	1.05×10^{32}	14.66	3	1.13×10^{18}	16.34	3	3.54×10^{33}	17.52	2	2.88×10^{34}
0	32.76	28	∞	31.23	23	∞	34.57	24	∞	36.98	25	∞
1×10^{-16}	17.66	12	1.71×10^{31}	12.88	2	8.47×10^{17}	19.41	6	3.62×10^{19}	18.88	4	1.30×10^{35}
1×10^{-12}	8.81	2	1.62×10^{27}	13.71	1	1.49×10^{28}	18.21	4	1.02×10^{30}	18.13	3	3.27×10^{30}
1×10^{-8}	15.25	9	4.97×10^{18}	12.83	1	7.73×10^{24}	16.54	1	2.32×10^{25}	17.24	1	9.55×10^{26}
1×10^{-5}	11.19	5	2.62×10^{14}	11.75	1	7.26×10^{18}	14.90	1	3.87×10^{22}	15.92	0	1.40×10^{23}
1×10^{-4}	12.51	7	6.67×10^{10}	10.76	0	1.08×10^{21}	13.33	0	1.82×10^{19}	17.39	2	9.39×10^{19}
1×10^{-3}	11.03	5	1.32×10^{9}	10.87	0	4.25×10^{19}	14.14	1	2.38×10^{20}	15.23	0	3.00×10^{20}
1×10^{-2}	8.75	3	6.14×10^{7}	11.37	1	5.51×10^{18}	12.96	0	6.34×10^{19}	14.45	1	6.99×10^{20}
1×10^{-1}	7.95	2	3.12×10^{5}	10.50	0	2.16×10^{16}	12.60	0	1.39×10^{18}	16.42	3	3.62×10^{20}

The experimental evaluation of the BoxCQP algorithm under controlled indefiniteness reveals a strong dependence of convergence behavior on the condition number of the modified matrix B. As expected, the algorithm demonstrates robust performance in well-conditioned scenarios, particularly when the smallest diagonal entry remains significantly positive (e.g., 1×10^{-2} or 1×10^{-1}). In these cases, the mean iteration counts remain low and convergence failures are rare or nonexistent across all dimensions tested.

However, as the matrix becomes increasingly ill-conditioned—especially around diagonal perturbations close to zero or slightly negative—the condition number grows rapidly, often exceeding 10^{25}, and in extreme cases (e.g., with zero diagonal values), becomes infinite. These configurations correspond to a substantial rise in both iteration count and failure rates. For instance, when the diagonal includes zero, the algorithm fails to converge in up to 28% of the runs for $n = 10$, and similar behavior is observed for higher dimensions. Even for small negative values (e.g., -1×10^{-3} or -1×10^{-4}), the algorithm begins to exhibit instability as the condition number increases.

Interestingly, there appears to be a zone of tolerance: small perturbations toward indefiniteness (especially around -1×10^{-5} to 1×10^{-8}) do not always lead to immediate failure. Instead, BoxCQP still converges in many cases, albeit with increased iteration counts. This suggests some resilience of the solver near the boundary of positive definiteness. Nevertheless, the results clearly highlight the algorithm's sensitivity to definiteness and condition number, emphasizing the importance of matrix conditioning in practical applications. Incorporating condition number estimation or preconditioning strategies could enhance solver stability and broaden the range of problems to which BoxCQP can be reliably applied.

As a closing remark, we should point out that out of the total 6400 experiments (16 perturbation levels $\times$ 100 runs $\times$ 4 dimension settings), the BoxCQP algorithm successfully converged in 6146, i.e., 96%, of the cases. The rest of 4% correspond to indefinite Hessians.

3. State-of-the-Art Convex Quadratic Programming Solutions

There exist several quadratic programming codes in the literature. We have chosen to compare with three of them, specifically with QPBOX, QLD, and QUACAN. These codes share several common features so that the comparison is both meaningful and fair. All codes are written in the same language (FORTRAN 77) so that different language overheads are eliminated. Also, they are written by leading experts in the field of quadratic programming, so that their quality is guaranteed. Notice also that all codes are specific to the problem, and not of general purpose nature and are distributed freely through the World Wide Web at the time of writing.

3.1. QPBOX

QPBOX [30] is a Fortran77 package for box-constrained quadratic programs developed in IMM in the Technical University of Denmark. The bound-constrained quadratic program is solved via a dual problem, which is the minimization of an unbounded, piecewise quadratic function. The dual problem involves a lower bound of λ_1, i.e, the smallest eigenvalue of a symmetric, positive matrix, and is solved by Newton iteration with line search.

3.2. QLD

This code [31] is available due to K.Schittkowski of the University of Bayreuth, Germany and is a modification of routines due to MJD Powell at the University of Cambridge. It is essentially an active set, interior-point method and supports general linear constraints too.

3.3. QUACAN

This algorithm combines conjugate gradients with gradient projection techniques, as the algorithm of Moré and Toraldo [32]. A new strategy for the decision of leaving the current face is introduced, making it possible to obtain finite convergence even for a singular Hessian and in the presence of dual degeneracy. QUACAN [33] is specialized for convex problems subject to simple bounds.

4. Experimental Results—Fortran Implementation

To verify the effectiveness of the proposed approach, we experimented with five different problem types, and measured cpu times to make a comparison possible. We have implemented BoxCQP in Fortan 77 and used a recent Intel processor with a Linux operating system and employed the suite of the GNU gfortran compiler.

In the subsections that follow, we describe in brief the different test problems used for the experiments, and report our results.

4.1. Random Problems

The first set of experiments treats randomly generated problems. We generate problems following the general guidelines of [32]. Specific details about creating these random problems are provided in the Appendix A.1.

For every random problem class, we have created Hessian matrices with three different condition numbers:

1. Using $n_{cond} = 0.1$ and hence, $\kappa_2(B) = 1.259$;
2. Using $n_{cond} = 1$ and hence, $\kappa_2(B) = 10$;
3. Using $n_{cond} = 5$ and hence, $\kappa_2(B) = 10^5$.

The results for the three variants of BoxCQP against the other quadratic codes for the classes (a), (b), and (c) and for three different condition numbers are shown in Tables 1, 2, and 3, respectively. In each table, alongside the execution times of the competing solvers, we include additional columns presenting their rankings for the specific case. These rankings serve to facilitate a more general interpretation and comparison of the results.

The results presented in Table 3 show that across all conditioning levels, computational time increases with problem size, as expected. Variant 1 (LDL^T decomposition) performs well for small-to-moderate problem sizes but suffers from scalability issues as the problem dimension increases, particularly in ill-conditioned cases where factorization becomes computationally expensive. In contrast, Variant 2 (conjugate gradient) exhibits greater robustness for ill-conditioned problems but is slower than direct decomposition for well-conditioned cases. The hybrid Variant 3 provides the best overall performance, maintaining lower runtimes across different problem sizes and conditioning levels.

Comparing BoxCQP to the antagonistic solvers, QUACAN performs well on small problems but becomes inefficient for large-scale and ill-conditioned cases. QPBOX and QLD exhibit competitive runtimes, with QLD showing the best efficiency in large, ill-conditioned scenarios. BoxCQP Variants 2 and 3 consistently outperform QUACAN, making them preferable for difficult optimization problems. In this case, BoxCQP Variant 3 emerges as the most effective approach, striking a balance between computational efficiency and robustness to ill-conditioning.

Considering the results presented in Tables 4 and 5, we can see that for the most ill-conditioned cases, Variant 1 (LDL) becomes prohibitively expensive, and methods using iterative methods (Variants 2 and 3) become more favorable. On the other hand, for the well-conditioned cases, iterative BoxCQP variants outperform the LDL one. Among the other solvers, QUACAN struggles significantly with ill-conditioned problems and large problem sizes, often displaying dramatically increased runtime, particularly for

$\kappa_2(B) = 10^5$. QPBOX and QLD scale more effectively, with QLD consistently outperforming QUACAN in all problem sizes. However, BoxCQP Variants 2 and 3 maintain superior performance over QUACAN and, in all cases, are better than QPBOX and QLD.

Table 3. Random table results; $act\_prob = 0.5$, $up\_low\_prob = 0.5$.

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD	Var.1	Var.2	Var.3	QN	QPB	QLD
Rand (ncond = 0.1, n = 200)	0.02	**0.00**	**0.00**	**0.00**	0.06	0.08	4	1	1	1	5	6
Rand (ncond = 1, n = 200)	0.02	**0.00**	0.01	0.03	0.05	0.07	3	1	2	4	5	6
Rand (ncond = 5, n = 200)	**0.03**	0.18	**0.03**	1.38	0.07	0.07	1	5	1	6	3	3
Rand (ncond = 0.1, n = 400)	0.19	**0.02**	0.05	0.06	0.44	0.60	4	1	2	3	5	6
Rand (ncond = 1, n = 400)	0.22	**0.08**	0.11	0.25	0.47	0.61	3	1	2	4	5	6
Rand (ncond = 5, n = 400)	**0.37**	2.00	0.42	16.16	0.69	0.58	1	5	2	6	4	3
Rand (ncond = 0.1, n = 600)	0.78	**0.06**	0.12	0.14	1.43	2.13	4	1	2	3	5	6
Rand (ncond = 1, n = 600)	0.90	**0.24**	0.37	0.66	1.57	2.14	4	1	2	3	5	6
Rand (ncond = 5, n = 600)	**1.16**	9.72	1.27	48.69	1.97	2.14	1	5	2	6	3	4
Rand (ncond = 0.1, n = 800)	2.58	**0.13**	0.27	0.31	3.80	5.44	4	1	2	3	5	6
Rand (ncond = 1, n = 800)	2.59	**0.42**	0.53	1.26	3.76	5.37	4	1	2	3	5	6
Rand (ncond = 5, n = 800)	**3.58**	32.21	3.72	108.13	5.76	5.47	1	5	2	6	4	3
Rand (ncond = 0.1, n = 1000)	4.52	**0.22**	0.54	0.50	7.83	10.17	4	1	3	2	5	6
Rand (ncond = 1, n = 1000)	4.58	**0.66**	0.95	1.68	7.93	10.00	4	1	2	3	5	6
Rand (ncond = 5, n = 1000)	**6.82**	72.48	8.19	143.93	10.02	9.93	1	5	2	6	4	3
Rand (ncond = 0.1, n = 1200)	9.40	**0.33**	1.23	0.75	13.26	18.05	4	1	3	2	5	6
Rand (ncond = 1, n = 1200)	9.02	**0.96**	2.29	3.32	13.49	19.19	4	1	2	3	5	6
Rand (ncond = 5, n = 1200)	11.08	90.54	**11.05**	187.50	16.90	19.31	2	5	1	6	3	4
Rand (ncond = 0.1, n = 1400)	12.03	**0.43**	1.57	1.14	20.94	30.16	4	1	3	2	5	6
Rand (ncond = 1, n = 1400)	11.97	**1.20**	2.15	3.51	21.41	30.57	4	1	2	3	5	6
Rand (ncond = 5, n = 1400)	**17.48**	118.56	17.92	300.58	27.72	30.07	1	5	2	6	3	4
Rand ncond = 0.1, n = 1600)	23.34	**0.61**	2.68	1.76	29.76	48.56	4	1	3	2	5	6
Rand ncond = 1, n = 1600)	26.29	**2.60**	4.57	7.79	33.50	46.80	4	1	2	3	5	6
Rand ncond = 5, n = 1600)	**36.36**	302.91	49.70	895.78	37.99	46.23	1	5	4	6	2	3
Rand ncond = 0.1, n = 1800)	29.42	**0.68**	4.43	2.13	43.82	64.54	4	1	3	2	5	6
Rand ncond = 1, n = 1800)	28.75	**1.93**	4.79	7.11	43.91	64.10	4	1	2	3	5	6
Rand ncond = 5, n = 1800)	**48.49**	352.77	57.25	925.31	57.65	63.13	1	5	2	6	3	4
Rand ncond = 0.1, n = 2000)	47.95	**0.92**	6.05	2.55	58.39	89.83	4	1	3	2	5	6
Rand ncond = 1, n = 2000)	47.75	**2.75**	7.32	7.57	60.00	91.10	4	1	2	3	5	6
Rand ncond = 5, n = 2000)	74.03	395.21	**70.79**	711.86	103.15	89.13	2	5	1	6	4	3
Average ranking							**3.00**	**2.33**	**2.13**	**3.80**	**4.43**	**5.13**

Overall, the results confirm that when most variables reside on the bounds, BoxCQP Variant 3 is the most practical choice, as it combines the advantages of both LDL^T and CG, adapting well to different problem conditions. Variant 1 remains suitable for well-conditioned small problems, whereas Variant 2 is preferable for handling ill-conditioned, large-scale problems.

The performance scaling plot in Figure 2 compares the runtime of three algorithmic variants for solving well-conditioned problems, where half of the variables are fixed on bounds. The primary computational task in the algorithm is solving a linear system at each iteration, and the three variants differ in their approach to this task. We can an infer that for small problem sizes, the differences between the three variants are relatively minor. However, as the problem size grows, Variant 1 (LDL^T decomposition at every iteration) exhibits the steepest runtime increase due to the high cost of repeated factorizations. Meanwhile, Variant 2 (CG throughout) shows much better scalability, with its runtime growing at a slower rate, making it preferable for large-scale problems. Variant 3 (hybrid) likely demonstrates an intermediate performance profile, outperforming Variant 1 in terms of efficiency while maintaining better numerical robustness than Variant 2.

In summary, if computational speed is the primary concern, Variant 2 (CG) is the most scalable option, particularly for large problem sizes. If numerical stability and accuracy are the priority, Variant 1 (LDL^T) is the most robust but at the cost of significantly higher compu-

tation time. Variant 3 (hybrid) emerges as an optimal middle-ground approach, balancing efficiency and stability by combining iterative and direct methods. The performance-scaling trends confirm these expectations, with Variant 1 becoming increasingly costly, Variant 2 scaling efficiently, and Variant 3 offering a competitive compromise.

Table 4. Random table results; $act\_prob = 0.9$, $up\_low\_prob = 0.5$.

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD	Var.1	Var.2	Var.3	QN	QPB	QLD
Rand (ncond = 0.1, n = 200)	0.02	**0.00**	**0.00**	0.01	0.06	0.10	4	1	1	3	5	6
Rand (ncond = 1, n = 200)	0.02	**0.01**	**0.01**	**0.01**	0.06	0.11	4	1	1	1	5	6
Rand (ncond = 5, n = 200)	0.08	0.10	**0.02**	2.20	0.06	0.10	3	4	1	6	2	4
Rand (ncond = 0.1, n = 400)	0.16	**0.02**	**0.02**	0.05	0.46	0.80	4	1	1	3	5	6
Rand (ncond = 1, n = 400)	0.16	**0.05**	**0.05**	0.11	0.44	0.82	4	1	1	3	5	6
Rand (ncond = 5, n = 400)	**0.19**	0.64	**0.19**	10.93	0.49	0.81	1	4	1	6	3	5
Rand (ncond = 0.1, n = 600)	0.70	**0.05**	**0.05**	0.15	1.59	2.93	4	1	1	3	5	6
Rand (ncond = 1, n = 600)	0.64	**0.13**	**0.13**	0.35	1.48	3.16	4	1	1	3	5	6
Rand (ncond = 5, n = 600)	**0.77**	2.29	0.78	35.44	1.57	2.99	1	4	2	6	3	5
Rand (ncond = 0.1, n = 800)	2.39	**0.11**	**0.11**	0.26	3.77	7.53	4	1	1	3	5	6
Rand (ncond = 1, n = 800)	2.42	0.35	**0.34**	0.96	3.82	7.47	4	2	1	3	5	6
Rand (ncond = 5, n = 800)	2.52	9.36	**2.10**	63.45	4.09	7.40	2	5	1	6	3	4
Rand (ncond = 1, n = 1000)	3.57	**0.43**	**0.43**	1.45	7.27	15.08	4	1	1	3	5	6
Rand (ncond = 5, n = 1000)	3.91	9.30	**3.07**	180.95	8.53	15.12	2	4	1	6	3	5
Rand (ncond = 0.1, n = 1200)	9.40	**0.32**	1.22	0.74	13.25	18.05	4	1	3	2	5	6
Rand (ncond = 1, n = 1200)	8.40	**0.73**	**0.73**	3.01	12.95	25.86	4	1	1	3	5	6
Rand (ncond = 5, n = 1200)	7.69	19.61	**5.51**	209.04	13.56	27.15	2	4	1	6	3	5
Rand (ncond = 0.1, n = 1400)	12.03	**0.43**	1.56	1.14	20.94	30.15	4	1	3	2	5	6
Rand (ncond = 1, n = 1400)	12.53	**1.29**	**1.29**	3.32	20.75	40.12	4	1	1	3	5	6
Rand (ncond = 5, n = 1400)	13.08	55.20	11.63	363.79	21.79	39.71	2	5	1	6	3	4
Rand (ncond = 0.1, n = 1600)	23.34	**0.61**	2.68	1.76	29.76	48.55	4	1	3	2	5	6
Rand (ncond = 1, n = 1600)	23.88	**2.06**	2.07	7.24	33.57	63.22	4	1	2	3	5	6
Rand (ncond = 5, n = 1600)	22.53	64.27	22.67	663.67	33.52	69.35	1	4	2	6	3	5
Rand (ncond = 0.1, n = 1800)	28.56	**0.61**	**0.61**	1.59	51	88.60	4	1	1	3	5	6
Rand (ncond = 1, n = 1800)	28.57	**1.65**	1.66	5.11	43.85	88.02	4	1	2	3	5	6
Rand (ncond = 5, n = 1800)	30.31	106.52	24.51	488.60	45.26	88.21	2	5	1	6	3	4
Rand (ncond = 0.1, n = 2000)	39.84	**0.59**	0.61	2.14	73.93	128.85	4	1	2	3	5	6
Rand (ncond = 1, n = 2000)	40.05	**2.36**	2.41	8.78	62.69	128.56	4	1	2	3	5	6
Rand (ncond = 5, n = 2000)	42.04	147.56	**32.04**	613.54	67.91	128.35	2	5	1	6	3	4
Average Ranking							3.24	2.21	1.41	3.86	4.28	5.48

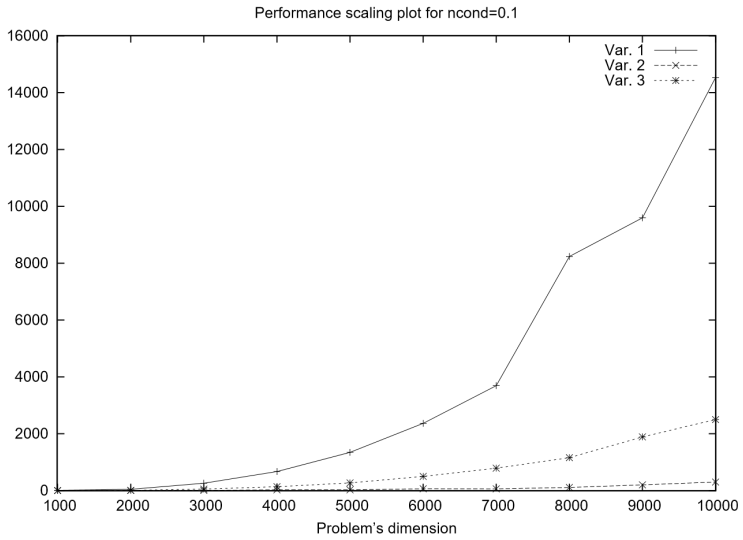


Figure 2. BoxCQP variants' scaling results for $ncond = 0.1, \kappa_2(B) = 1.259$.

Table 5. Random table results; $act\_prob = 0.1$, $up\_low\_prob = 0.5$.

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD	Var.1	Var.2	Var.3	QN	QPB	QLD
Rand (ncond = 0.1, n = 200)	0.03	**0.00**	0.02	0.01	0.06	0.03	4	1	3	2	6	4
Rand (ncond = 0.1, n = 400)	0.03	**0.01**	0.02	0.03	0.06	0.03	3	1	2	3	6	3
Rand (ncond = 0.1, n = 600)	0.06	0.82	0.10	1.00	0.07	**0.04**	2	5	4	6	3	1
Rand (ncond = 0.1, n = 800)	0.28	**0.03**	0.14	0.05	0.52	0.30	4	1	3	2	6	5
Rand (ncond = 0.1, n = 1000)	0.29	**0.10**	0.19	0.20	0.51	0.29	4	1	2	3	6	4
Rand (ncond = 0.1, n = 1200)	0.73	13.43	1.15	11.81	0.57	**0.29**	3	6	4	5	2	1
Rand (ncond = 0.1, n = 1400)	1.12	**0.08**	0.52	0.12	1.70	1.03	5	1	3	2	6	4
Rand (ncond = 0.1, n = 1600)	1.08	**0.24**	0.59	0.55	1.71	1.07	5	1	3	2	6	4
Rand (ncond = 0.1, n = 1800)	2.69	36.03	3.69	37.16	1.84	**1.05**	3	5	4	6	2	1
Rand (ncond = 0.1, n = 2000)	3.96	**0.20**	1.67	0.31	4.58	2.97	5	1	3	2	6	4
Rand (ncond = 1, n = 200)	4.18	**0.60**	2.08	1.13	4.69	2.73	5	1	3	2	6	4
Rand (ncond = 1, n = 400)	9.41	113.31	13.11	84.98	5.02	**2.87**	3	6	4	5	2	1
Rand (ncond = 1, n = 600)	8.10	**0.33**	4.12	0.49	9.67	4.92	5	1	3	2	6	4
Rand (ncond = 1, n = 800)	7.80	**0.93**	4.17	1.99	9.58	5.23	5	1	3	2	6	4
Rand (ncond = 1, n = 1000)	21.76	150.66	23.29	137.16	10.21	**4.78**	3	6	4	5	2	1
Rand (ncond = 1, n = 1200)	15.27	**0.50**	7.10	0.66	15.82	9.64	5	1	3	2	6	4
Rand (ncond = 1, n = 1400)	15.40	**1.42**	7.73	3.18	15.69	9.55	5	1	3	2	6	4
Rand (ncond = 1, n = 1600)	38.69	332.99	51.92	208.28	16.41	**9.46**	3	6	4	5	2	1
Rand (ncond = 1, n = 1800)	24.19	**0.69**	12.07	1.28	25.21	14.47	5	1	3	2	6	4
Rand (ncond = 1, n = 2000)	24.13	**1.92**	12.59	3.87	25.08	14.61	5	1	3	2	6	4
Rand (ncond = 0.1, n = 1600)	51.78	570.86	93.98	254.72	26.55	**14.44**	3	6	4	5	2	1
Rand (ncond = 5, n = 400)	42.09	**1.09**	18.83	2.06	39.03	25.60	6	1	3	2	5	4
Rand (ncond = 5, n = 600)	60.71	**4.21**	38.91	8.60	39.03	25.60	6	1	4	2	5	3
Rand (ncond = 5, n = 800)	80.80	482.62	109.59	765.19	41.66	**25.64**	3	5	4	6	2	1
Rand (ncond = 5, n = 1000)	56.08	**1.13**	28.06	1.67	51.92	33.74	6	1	3	2	5	4
Rand (ncond = 5, n = 1200)	54.58	**3.15**	27.59	6.78	52.20	33.69	6	1	3	2	5	4
Rand (ncond = 5, n = 1400)	130.07	819.08	130.85	876.60	55.17	**34.36**	3	5	4	6	2	1
Rand (ncond = 5, n = 1600)	78.98	**1.36**	37.10	1.91	69.26	49.50	6	1	3	2	5	4
Rand (ncond = 5, n = 1800)	114.84	**4.57**	74.21	8.00	69.67	49.85	6	1	5	2	4	3
Rand (ncond = 5, n = 2000)	189.95	921.16	269.69	670.03	73.12	**50.21**	3	6	4	5	2	1
Average Ranking							4.33	2.53	3.37	3.20	4.47	2.93

For ill-conditioned problems (see Figure 3, the performance-scaling behavior of the three algorithmic variants shifts significantly compared to well-conditioned problems. In such cases, the system matrix exhibits a high condition number, which impacts both direct and iterative solution methods differently.

From the runtime trends observed in Figure 3 in the new performance scaling plot, it appears that Variant 1 (LDL^T) performs better than in the well-conditioned scenario. This is likely due to the fact that LDL^T decomposition, being a direct method, remains robust even when the system matrix is poorly conditioned. Unlike iterative methods, which suffer from slow convergence or numerical instability when the condition number is large, LDL^T maintains accuracy at the cost of higher computational effort per iteration. However, in ill-conditioned problems, iterative solvers such as the conjugate gradient (CG) method (used in Variant 2) may require significantly more iterations to converge, making them less efficient overall. This explains why Variant 1, despite its higher theoretical complexity, outperforms the other two methods in this setting.

Variant 3 balances these trade-offs effectively by leveraging the fast initial approximations of CG while maintaining the stability of LDL^T in subsequent iterations. While CG may struggle with slow convergence in ill-conditioned problems, using it only in the first iteration can still provide a useful initial guess that reduces the effort needed for LDL^T decomposition later on. This reduces the total computational burden of LDL^T while still ensuring that subsequent iterations remain numerically stable.

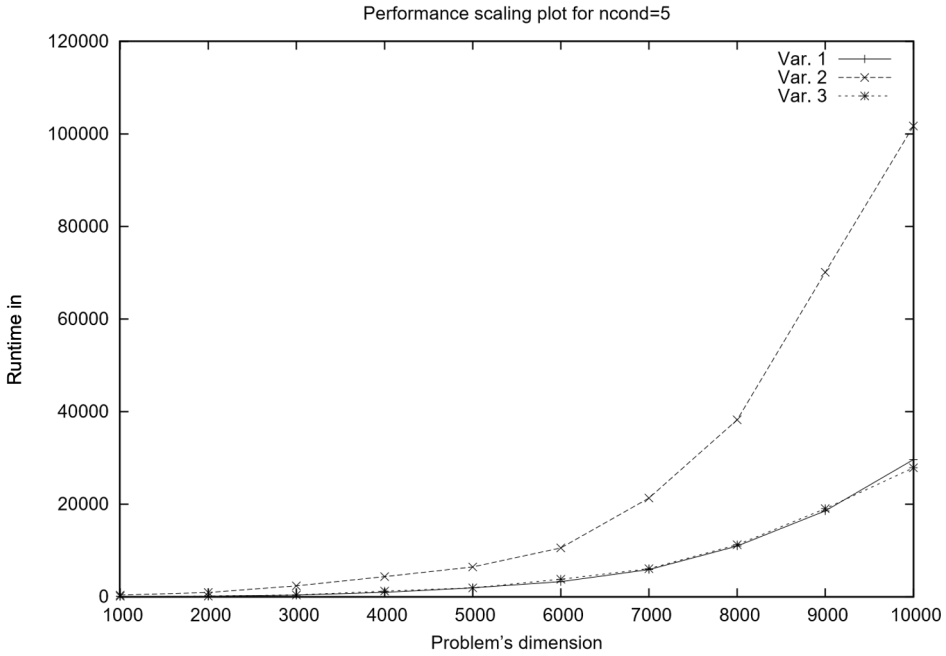


Figure 3. BoxCQP variants' scaling results for $ncond = 5, \kappa_2(B) = 10^5$.

4.2. Circus-Tent Problem

The circus-tent problem serves as a foundational case that highlights how mathematical optimization techniques can be applied to real-world engineering challenges. Its formulation as box-constrained quadratic programming problem is described in detail in Appendix A.2. Table 6 presents execution times for different solution approaches to the circus-tent problem across increasing problem sizes, measured in seconds. As the problem size grows, Variant 1 becomes increasingly expensive due to the repeated direct factorization, reaching 1333.51 s for $n = 4900$. Variant 2, relying entirely on the iterative CG method, exhibits significantly better scalability, with execution time increasing more gradually to 150.49 s at $n = 4900$. Variant 3, which blends iterative and direct methods, initially performs better than Variant 1 but eventually shows similar growth in execution time, reaching 696.21 s for the largest problem size. Among the solvers compared, QUACAN successfully solves only the smallest in negligible time but fails for larger cases. QPBOX is unable to solve any instance, as indicated by the NC values across all entries. QLD, however, demonstrates competitive performance, outperforming Variant 1 and Variant 3 for larger problems, reaching 617.58 s for $n = 4900$.

Table 6. Results for circus-tent problem case (single-thread execution time in seconds; NC stands for no convergence).

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD	Var.1	Var.2	Var.3	QN	QPB	QLD
Tent (n = 100)	0.01	**0.00**	**0.00**	0.00	NC	**0.00**	5	1	1	1	6	1
Tent (n = 400)	0.32	**0.13**	0.21	NC	NC	0.25	5	1	2	6	6	3
Tent (n = 900)	5.57	**1.45**	3.27	NC	NC	2.77	5	1	3	6	6	2
Tent (n = 1600)	48.30	**9.57**	29.11	NC	NC	20.53	5	1	3	6	6	2
Tent (n = 3600)	557.74	**55.74**	284.05	NC	NC	246.04	5	1	3	6	6	2
Tent (n = 4900)	1333.51	**150.49**	696.21	NC	NC	617.58	5	1	3	6	6	2
Average Ranking							5.00	1.00	2.83	6.00	6.00	2.00

From these results, Variant 2 (conjugate gradient method) proves to be the most efficient and scalable, particularly for larger problem sizes, making it the preferred choice for

large-scale computations. Variant 3 (hybrid approach) offers a reasonable compromise between iterative and direct methods, performing well for small-to-medium problems before exhibiting similar computational demands to Variant 1. Variant 1 (LDL^T decomposition), while accurate, is computationally expensive and less suitable for large-scale applications. Among the solvers, QLD remains the most competitive alternative, performing consistently better than LDL^T-based methods for larger problem sizes.

4.3. Biharmonic Equation Problem

The biharmonic equation arises in elasticity theory and describes the small vertical deformations of a thin elastic membrane. In this work, we consider an elastic membrane clamped on a rectangular boundary and subject to a vertical force while being constrained to remain below a given obstacle. This leads to a constrained variational problem that can be formulated as a convex quadratic programming (QP) problem with bound constraints. For more details about the formulation, see Appendix A.3.

Table 7 presents execution times (in seconds) for different numerical methods applied to the biharmonic equation problem across increasing problem sizes. As the problem size increases, Variant 1 exhibits the steepest growth in execution time, reaching 816.13 s for the largest case ($n = 4900$). Variant 2, which leverages iterative solvers, scales better, achieving a significantly lower execution time of 705.52 s for the same problem size. Variant 3 strikes a balance between direct and iterative methods, showing better performance than Variant 1 but remaining slightly less efficient than Variant 2, with 484.45 s for the case $n = 4900$.

In the series of solver tests, QLD demonstrates superior performance compared to QUACAN and QPBOX, solving the largest problem in 1837.66 s. By contrast, QUACAN and QPBOX exhibit considerably worse scalability, with times of 8282.04 s and 3067.21 s, respectively, for $n = 4900$. QUACAN shows effectiveness with smaller problems but becomes very inefficient as the problem size grows, whereas QPBOX follows a similar pattern, though it performs somewhat better. The findings emphasize that Variant 3 is the most scalable method, making it the optimal choice for large-scale biharmonic problems. Additionally, all BoxCQP variants outperform the competition in this scenario.

Table 7. Results for the biharmonic equation case (single-thread execution time in seconds).

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD	Var.1	Var.2	Var.3	QN	QPB	QLD
Biharm (n = 100)	**0.00**	**0.00**	**0.00**	**0.00**	0.01	0.01	1	1	1	1	5	5
Biharm (n = 400)	0.20	0.21	**0.19**	0.65	0.64	0.75	2	3	1	5	4	6
Biharm (n = 900)	4.28	3.13	**2.92**	18.52	10.49	9.29	3	2	1	6	5	4
Biharm (n = 1600)	23.33	17.89	**15.31**	119.66	82.12	60.87	3	2	1	6	5	4
Biharm (n = 2500)	106.19	77.24	**60.57**	775.03	333.91	222.79	3	2	1	6	5	4
Biharm (n = 3600)	308.73	271.46	**186.89**	2988.08	1071.34	684.57	3	2	1	6	5	4
Biharm (n = 4900)	816.13	705.52	**484.45**	8282.04	3067.21	1837.66	3	2	1	6	5	4
Average Ranking							2.57	2.00	1.00	5.14	4.86	4.43

4.4. Intensity-Modulated Radiation Therapy

Intensity-Modulated Radiation Therapy (IMRT) is an advanced radiotherapy technique that optimizes the spatial distribution of radiation to maximize tumor control while minimizing damage to surrounding healthy tissues and vital organs. The goal is to deliver a precisely calculated radiation dose that conforms to the tumor shape, reducing side effects and improving treatment effectiveness.

This problem is typically formulated as a quadratic programming (QP) task, where the objective is to determine the optimal fluence intensity profile for a given set of beam

configurations. The radiation dose distribution can be represented as a linear combination of beamlet intensities, allowing for a mathematical optimization approach. Given a set of desired dose levels, the optimal beamlet intensities are computed by solving a quadratic objective function that minimizes the difference between the prescribed and delivered doses. The optimization constraints include dose limits for critical organs and physical feasibility conditions. In Appendix A.4, we present in some details the derivation of the formulation.

In practical applications, inverse treatment planning in IMRT requires solving a quadratic optimization problem of the form shown in Equation (9) multiple times, as beam configurations are iteratively adjusted to meet clinical constraints. Since the process involves large-scale quadratic systems, efficient solvers are essential to ensure fast and accurate treatment planning.

$$\min_f s(f) = \frac{1}{2} f^T A f + f^T b \quad \text{subject to } f \geq 0 \tag{9}$$

Table 8 showcases findings derived from actual data generously shared by S. Breedveld [34]. In this scenario, seven beams are integrated, leading to a quadratic problem comprising 2342 parameters. In this instance, Variants 2 and 3 exhibit superior performance, outperforming their counterparts by a factor of two.

Table 8. Results for the Intensity-Modulated Radiation Therapy case (single-thread execution time in seconds).

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD	Var.1	Var.2	Var.3	QN	QPB	QLD
IMRT (n = 2342)	54.22	**33.11**	40.56	85.11	67.88	73.22	3	1	2	6	4	5
Average Ranking							3.00	1.00	2.00	6.00	4.00	5.00

4.5. Support Vector Classification

To create the problem for the case of Support Vector Classification, we used the CLOUDS dataset [35] a well-estabalished two dimensional and two-class classification task. The specifics of the quadratic programming formulation are provided in the Appendix A.5. We run different experiments for an increasing number of CLOUDS datapoints. From Table 9, we can see that Variant 1 experiences significant growth in execution time, reaching 263.35 s for $n = 3000$, making it the least efficient among the three variants. In contrast, Variant 2 scales much better due to its reliance on iterative methods, requiring 63.97 s for the largest problem. Variant 3, which combines both iterative and direct methods, shows even better performance than Variant 1 for larger problems, reducing execution time to 151.40 s at . When comparing solver performance, QUACAN, QPBOX, and QLD also display increasing execution times as problem sizes grow. QUACAN shows relatively poor scalability, requiring 1068.97 s for $n = 3000$, making it the slowest solver in the test. QPBOX, while performing better, still struggles with larger problem sizes, reaching 264.56 s at $n = 3000$. QLD, is completing the largest problem in 354.43 s, yet still significantly slower than Variants 2 and 3. These results highlight that Variant 2 (conjugate gradient method) is the most scalable approach, making it the preferred choice for large-scale SVM problems. Variant 3 (hybrid approach) still offers the best balance between iterative and direct solvers. Therefore, the findings suggest that a combination of CG-based techniques is optimal for solving large-scale SVM optimization problems efficiently.

Table 9. Results for SVM training (single-thread execution time in seconds).

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD	Var.1	Var.2	Var.3	QN	QPB	QLD
SVM (n = 100)	**0.00**	**0.00**	0.00	0.01	0.01	**0.00**	1	1	1	5	5	1
SVM (n = 200)	**0.04**	**0.04**	**0.04**	0.15	0.07	0.07	3	1	1	6	4	4
SVM (n = 300)	0.14	**0.09**	0.10	0.39	0.23	0.25	3	1	2	6	4	5
SVM (n = 400)	0.37	**0.29**	0.33	2.09	0.53	0.63	3	1	2	6	4	5
SVM (n = 500)	0.70	**0.61**	0.71	4.69	1.05	1.27	2	1	3	6	4	5
SVM (n = 600)	1.18	**0.84**	0.97	6.54	1.85	2.36	3	1	2	6	4	5
SVM (n = 700)	2.26	**1.51**	1.85	14.33	3.03	3.81	3	1	2	6	4	5
SVM (n = 800)	3.38	**1.87**	2.25	21.75	4.68	6.19	3	1	2	6	4	5
SVM (n = 900)	5.67	**3.40**	3.84	27.16	7.33	8.20	3	1	2	6	4	5
SVM (n = 1000)	7.26	**4.29**	5.01	34.01	10.39	11.33	3	1	2	6	4	5
SVM (n = 2000)	68.78	**22.01**	36.24	256.22	77.29	104.31	3	1	2	6	4	5
SVM (n = 3000)	263.35	**63.97**	151.40	1068.97	264.56	354.43	3	1	2	6	4	5
Average Ranking							2.75	1.00	1.92	5.92	4.08	4.58

4.6. Summarizing Fortran Experimental Results

The performance of the six solver variants—Variant 1, Variant 2, Variant 3 versus QUACAN, QPBOX, and QLD—was evaluated across a diverse set of convex quadratic programming problems.

Each case revealed different characteristics of the solver behavior. In the random problem set, Variant 3 and Variant 2 consistently achieved the lowest average rankings (2.30 and 2.35, respectively, on all 90 cases), indicating robust and efficient performance, especially under varying condition numbers. QUACAN followed with an average rank of 3.86, while QPBOX and QLD were generally slower (4.40 and 5.53, respectively), reflecting their higher computational overheads.

In the context of the biharmonic scenario, characterized by structured sparse problems, Variant 3 emerged as the leader, boasting the top average rank (1.00), with Variant 2 not far behind (2.00). A similar pattern occurred in the circus-tent problem, where Variant 2 was clearly the standout performer, achieving the highest average rank (1.00), while QLD followed in second place (2.00), demonstrating its effectiveness for structured geometric problems when feasible. For SVM classification tasks, Variant 2 excelled, with an average rank of 1.00, and was trailed by Variant 3 and Variant 1. Meanwhile, QLD and QPBOX showed inferior performance, especially with large datasets, due to scaling challenges. In the singular IMRT instance, Variant 2 along with Variant 3 were rated highest, emphasizing their competitiveness even in substantial real-world applications.

In Table 10 we present aggregated ranking results for all the test problem cases. When evaluating all problems together, the aggregate average ranking supports the prior findings. Variant 3 and Variant 2 secured top overall rankings of 2.11 and 2.33, respectively, with Var.1 following at 3.00. Among the other solvers, QUACAN occupied a central rank of 3.81, while QPBOX and QLD frequently ranked lower at 4.44 and 5.11, respectively. These outcomes highlight the robustness and versatility of the proposed BoxCQP variants, particularly Var.2 and Var.3, which effectively balance speed and dependability across diverse problem categories.

Table 10. Aggregated results for all cases (average ranking).

Prob. Name	Var.1	Var.2	Var.3	QN	QPB	QLD
Total Average Ranking	3.00	2.33	**2.11**	3.81	4.44	5.11

5. Experimental Results—Python Implementation

Porting code from Fortran 77 (F77) and MATLAB R2023b to Python 3.8 offers significant advantages in terms of accessibility, reproducibility, and community engagement,

making it highly beneficial for exposure in the scientific computing community. Python's open-source nature, extensive ecosystem, interoperability, and modern programming features make it an ideal platform for sharing, optimizing, and scaling scientific applications. In this section, we present some experimental results from the Python implementation of BoxCQP.

5.1. Linear Least Squares with Bound Constraints

Linear least squares with bound constraints (LLSBC) is a convex optimization problem that bridges classical linear algebra and modern optimization theory. By imposing simple bound constraints to a linear least-squares objective, we obtain a quadratic program (QP) that is guaranteed to be convex.

In practice, adding bound constraints to a least-squares problem is hugely important because it lets us incorporate prior knowledge or physical limitations into the solution. Ordinary linear least squares (OLS) may produce solutions that are mathematically optimal but physically impossible or undesirable (e.g., negative values for inherently non-negative quantities, or parameters outside a feasible range). LLSBC addresses this by enforcing simple "box" constraints ($\ell_i \leq x_i \leq u_i$) on the solution vector. This leads to more realistic and interpretable models in many fields. For instance, in machine learning and statistics, it is often unreasonable for certain coefficients to be negative or to exceed certain values (consider probabilities, ages, or concentrations). By constraining coefficients to be non-negative or within a plausible range, we guarantee that the model's outputs make sense (e.g., predicted prices or counts cannot be negative)

In engineering and the sciences, bound constraints allow us to respect physical laws or design limits—for example, in control systems, we might require gain parameters to remain within stable ranges, or in curve fitting, we might enforce that a response is non-decreasing with an input. In signal processing and image processing, constraints like non-negativity (pixel intensities, power spectra) or monotonicity can significantly improve solutions by reducing noise artifacts and preventing unphysical oscillations. Overall, LLSBC is interesting because it enhances the least-squares approach with robustness and domain knowledge, making the solutions applicable in real-world scenarios where unconstrained solutions would fail. It strikes a useful balance: retaining the computational efficiency and well-understood nature of least squares, while adding just enough constraints to capture practical requirements.

Many regression and estimation tasks in machine learning benefit from bound constraints. A notable example is non-negative least squares (NNLS), where we require $x_i \geq 0$ for all i. This is used when model coefficients represent quantities that cannot go below zero. For instance, when fitting a model to predict prices, ages, or counts, allowing negative coefficients or predictions is not meaningful Imposing non-negativity yields more sensible models and can also have a regularizing effect (often promoting sparsity in the solution similar to an L_1 penalty). NNLS is widely used as a subroutine in matrix factorization problems like PARAFAC and non-negative matrix factorization (NMF), where one alternates solving least-squares subproblems under non-negativity constraints. This helps extract interpretable features (e.g., in text mining, image analysis, or clustering) because each factor is constrained to contribute additively (no negative cancellations). Another application is isotonic regression, which is a least-squares problem with a monotonicity constraint ($x_1 \leq x_2 \leq \cdots \leq x_n$). This can be formulated as LLSBC by introducing linear inequality constraints between variables.

5.2. Problem Definition (Bounded Least Squares)

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} \ \frac{1}{2}\,\|Ax-b\|_2^2, \\ &\text{subject to } \ \ell_i \ \le \ x_i \ \le \ u_i, \quad i=1,\ldots,n\,. \end{aligned} \tag{10}$$

where:

- A is an $m \times n$ design matrix;
- b is an m-vector of observations;
- ℓ and u are n-vectors (or scalars) defining lower and upper bounds.

5.3. Expanding the Objective Function

$$\frac{1}{2}\|Ax-b\|_2^2 = \frac{1}{2}(Ax-b)^T(Ax-b). \tag{11}$$

Expanding this quadratic term:

$$\frac{1}{2}(Ax-b)^T(Ax-b) = \frac{1}{2}x^TA^TAx - b^TAx + \frac{1}{2}b^Tb. \tag{12}$$

Ignoring the constant term $\frac{1}{2}b^Tb$ (since it does not affect the minimizer), we obtain the standard **convex quadratic program (QP) form**:

$$\min_x \ \frac{1}{2}x^T(A^TA)x - (A^Tb)^Tx, \quad \text{s.t. } \ell \le x \le u. \tag{13}$$

5.4. Standard Quadratic Programming Representation

Rewriting the problem in standard QP form:

$$\min_x \ \frac{1}{2}x^TQx + c^Tx, \quad \text{subject to } \ell \le x \le u, \tag{14}$$

where:

$$Q = A^TA \quad \text{(symmetric positive semidefinite matrix)}, \tag{15}$$

$$c = -A^Tb. \tag{16}$$

This formulation ensures that the problem is a convex QP since Q is positive semidefinite (or positive definite if A has full column rank).

Experimental Setup

For comparison, we utilize SciPy, a robust open-source Python library for scientific computing and optimization, offering efficient numerical methods for linear algebra, optimization, signal processing, and statistical analysis. Specifically, the scipy.optimize.lsq_linear function is employed to solve bounded linear least-squares problems of the form of Equation (10). The solution is obtained using two distinct approaches: Trust-Region Reflective (TRF) and Bounded Variable Least Squares (BVLS) algorithms.

The *TRF* (Trust-Region Reflective) algorithm [36] is a subspace trust-region method that is particularly effective for solving large-scale and well-conditioned least-squares problems. It operates by iteratively refining the solution within a trust region, ensuring that the step size remains appropriate to maintain stability. This method enforces bound constraints using an active-set strategy, meaning it considers only variables that are likely to be active at the optimal solution. Trust-region methods are robust, particularly when

dealing with ill-conditioned problems, as they naturally handle numerical instabilities and provide controlled step updates.

The *BVLS* (Bounded Variable Least Squares) algorithm [37] is a projection-based method that explicitly enforces bound constraints at each iteration. Unlike *TRF*, which works in a trust-region framework, *BVLS* solves a sequence of unconstrained least-squares problems while ensuring that the solution stays within the prescribed bounds. It follows a gradient-based active-set approach, where variables are either held at their bounds or updated according to the gradient direction, leading to the efficient handling of constraints. This method is particularly useful when strict bound enforcement is crucial, as it prevents overshooting beyond limits at any step.

Our Python implementation is presented in the Appendix A and it directly solves the problem in Equation (14). Notice that we include in timing the multiplications in Equations (15) and (16).

In Table 11 we present execution times (in seconds) for solving random linear least squares (LSQ) problems with bound constraints using three different methods: Variant 1, lsq_linear-TRF, and lsq_linear-BVLS. The problem sizes n vary in terms of the number of observations m. Variant 1 consistently outperforms both lsq_linear-TRF and lsq_linear-BVLS in terms of execution time. For smaller problem sizes, (such as $m = 2000$ and $n = 500$), Variant 1 completes the computation in 0.02 s, whereas lsq_linear-TRF and lsq_linear-BVLS require 0.95 s and 0.63 s, respectively. As the problem size increases, the execution time of Variant 1 scales more efficiently compared to the other two methods. For example, for dimension (m = 20,000, $n = 2000$), Variant 1 takes 1.26 s, whereas lsq_linear-TRF and lsq_linear-BVLS require 69.12 s and 49.71 s, respectively. This significant difference highlights the computational efficiency of Variant 1, particularly for large-scale problems.

Table 11. Linear least-squares random cases (single-thread execution time in seconds).

Prob. Name	Variant 1	lsq_linear-TRF	lsq_linear-BVLS
LSQ (m = 2000, n = 500)	**0.02**	0.95	0.63
LSQ (m = 2000, n = 1000)	**0.12**	4.22	2.23
LSQ (m = 2000, n = 1500)	**0.21**	6.85	12.30
LSQ (m = 2000, n = 2000)	**0.45**	12.45	25.37
LSQ (m = 20,000, n = 500)	**0.00**	4.75	3.42
LSQ (m = 20,000, n = 1000)	**0.28**	18.32	10.94
LSQ (m = 20,000, n = 1500)	**0.69**	35.12	25.83
LSQ (m = 20,000, n = 2000)	**1.26**	69.12	49.71

5.5. 225-Asset Portfolio Optimization Problem

The 225-Asset problem refers to a large-scale quadratic programming formulation used in portfolio optimization, where the objective is to minimize the portfolio's risk (variance) subject to a set of linear constraints. The problem involves a universe of 225 assets and is based on a classic mean-variance optimization model proposed by Markowitz [38].

Let $x \in \mathbb{R}^{225}$ be the portfolio weights vector, where each x_i represents the fraction of the total investment allocated to asset i. The problem is formulated as:

$$\begin{aligned} \min_{x \in \mathbb{R}^{225}} \quad & \frac{1}{2} x^\top \Sigma x \\ \text{subject to} \quad & r^\top x \geq r_{\min} \\ & \mathbf{1}^\top x = 1 \\ & x \geq 0 \end{aligned}$$

where

- $\Sigma \in \mathbb{R}^{225\times 225}$ is the positive definite covariance matrix of asset returns.
- $r \in \mathbb{R}^{225}$ is the expected return vector.
- $r_{\min} \in \mathbb{R}$ is the minimum required portfolio return.
- $\mathbf{1} \in \mathbb{R}^{225}$ is the vector of ones (to enforce full investment).

The constraints ensure:

- A minimum return threshold is met ($r^\top x \geq r_{\min}$).
- The portfolio is fully invested ($\mathbf{1}^\top x = 1$).
- No short selling is allowed ($x_i \geq 0\ \forall i$).

This problem is often used as a benchmark in quadratic programming solvers because of its size (225 variables and several hundred constraints) and its relevance in financial optimization. It tests a solver's ability to handle large, sparse quadratic programs with both inequality and equality constraints in practical applications. To express this problem in the standard quadratic programming form with inequality constraints of the form seen in Equation (2), we perform the following transformations:

1. Set $B = \Sigma$, and $d = 0$. There is no linear cost term in the mean-variance objective, only the quadratic risk term.
2. Encode the return constraint $r^\top x \geq r_{\min}$ as one row in A and b:
   $A_1 = r^\top$, $b_1 = r_{\min}$
3. Convert the equality constraint $\mathbf{1}^\top x = 1$ into two inequalities:
   $\mathbf{1}^\top x \geq 1$ and $-\mathbf{1}^\top x \geq -1$. These become rows in A:
   $A_2 = \mathbf{1}^\top$, $b_2 = 1$;
   $A_3 = -\mathbf{1}^\top$, $b_3 = -1$
4. Express the no short-selling constraint $x \geq 0$ as $A_4 = I$, the identity matrix, and $b_4 = 0$

Putting all constraints together:

$$A = \begin{bmatrix} r^\top \\ \mathbf{1}^\top \\ -\mathbf{1}^\top \\ I \end{bmatrix}, \quad b = \begin{bmatrix} r_{\min} \\ 1 \\ -1 \\ 0 \end{bmatrix}$$

This formulation is now fully compatible with solvers that accept inequality-only quadratic programs, such as those in the form of Equation (2).

To benchmark the Python implementation of BoxCQP, we compared its performance against two well-established quadratic programming solvers available through the `qpsolvers` Python interface: `quadprog` and `osqp` [39,40]. The `quadprog` solver is based on the Goldfarb–Idnani active-set method, which is particularly suitable for small- to medium-scale convex QP problems with dense Hessians. In contrast, `osqp` (Operator Splitting Quadratic Program) is a modern, operator-splitting-based solver that handles large-scale problems efficiently, even when the matrices involved are sparse or poorly conditioned. In Table 12, we present some preliminary results comparing BoxCQP to modern antagonistic methods.

Table 12. 225-Asset portfolio optimization (single-thread execution time in seconds).

Prob. Name	Variant 1	quadprog	osqp
225-Asset (m = 453, n = 225)	**0.0062**	0.0421	0.0091

5.6. Bound-Constrained Non-Linear Optimization

We introduce a trust-region method for non-linear optimization with bound constraints, where the trust region is defined as a hyperbox, differing from the conventional hypersphere or hyperellipsoid approaches. The rectangular trust region is particularly well suited for problems with bound constraints, as it maintains its geometric structure even when intersecting the feasible region.

Trust-region methods fall in the category of sequential quadratic programming [41,42]. These algorithms are iterative and the objective function $f(x)$ (assumed to be twice continuously differentiable) is approximated in a proper neighborhood of the current iterate (the trust region), using a quadratic model. Namely, at the k^{th} iteration, the model is given by:

$$f(x^k + s) \approx m^{(k)}(s) = f(x^{(k)}) + s^T g^{(k)} + \frac{1}{2} s^T B^{(k)} s \tag{17}$$

where $g^{(k)} = \nabla f(x^{(k)})$ and $B^{(k)}$, in the case of Newton's method, is a positive definite modification of the Hessian, while in the case of quasi-Newton methods, it is a positive definite matrix produced by the relevant update.

The trust region may be defined by:

$$\mathbf{T}^{(k)} = \{x \in \Re^n \mid ||x - x^{(k)}|| \leq \Delta^{(k)}\} \tag{18}$$

It is obvious that different choices for the norm lead to different trust-region shapes. The Euclidean norm $||\cdot||_2$ corresponds to a hypersphere, while the $||\cdot||_\infty$ norm defines a hyperbox.

Given the model and the trust region, we seek a step $||s^{(k)}|| \leq \Delta^{(k)}$ that minimizes $m^{(k)}(s)$. We compare the actual reduction $\delta f^{(k)} = f(x^{(k)}) - f(x^{(k)} + s^{(k)})$ to the model reduction $\delta m^{(k)} = m^{(k)}(0) - m^{(k)}(s^{(k)})$. If they agree to a certain extend, the step is accepted and the trust region is either expanded or remains the same. Otherwise, the step is rejected and the trust region is contracted. The basic trust-region algorithm is sketched in Algorithm 2.

Algorithm 2 Basic trust region

1. Pick the initial point and trust-region parameter $x^{(0)}$ and $\Delta^{(0)}$, and set $k = 0$.
2. Construct a quadratic model:

$$f(x^k + s) \approx m^{(k)}(s) = f(x^{(k)}) + s^T g^{(k)} + \frac{1}{2} s^T B^{(k)} s$$

3. Minimize $m^{(k)}(s)$ and hence determine $||s^{(k)}|| \leq \Delta^{(k)}$
4. Compute the ratio of actual to expected reduction: $r^{(k)} = \frac{\delta f^{(k)}}{\delta m^{(k)}}$, and update the trust region, following the strategy of Dennis and Schnabel [43] (Appendix A, page 338).
5. Increment $k \leftarrow k + 1$ and repeat from 1.

Consider the bound-constrained problem:

$$\min_x f(x), \quad \text{subject to: } l_i \leq x_i \leq u_i$$

The unconstrained case is obtained by letting $u_i = -l_i \to \infty$.
Let $x^{(k)}$ be the k-th iterate of the trust-region algorithm.
Hence, step 3 of Algorithm 2 becomes:

$$\min_{s} \; m^{(k)}(s) = s^T g^{(k)} + \frac{1}{2} s^T B^{(k)} s \tag{19}$$
$$\text{subject to: } \max(l_i - x_i^{(k)}, -\Delta^{(k)}) \leq s_i \leq \min(u_i - x_i^{(k)}, \Delta^{(k)})$$

We have developed a hybrid trust-region algorithm that utilizes the Hessian matrix to determine the appropriate optimization approach. When the Hessian is positive definite, the algorithm transitions to solving the quadratic subproblem (see Equation (20)). However, if the Hessian is indefinite, we employ the classical method described in [41] to ensure stability and convergence. It is important to note that in the vicinity of a local minimum, the Hessian matrix is typically positive definite, reinforcing the effectiveness of this approach. The complete implementation of the algorithm is provided in Appendix A Listing A3: Python TrustBox Implementation. Preliminary results with random settings (bounded and unbounded) of the well known Rosenbrock function

$$f(x) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \right], \text{ with minimum at } x^* = [1, 1, \ldots, 1]^T.$$

showed us that (a) half of the iterations are positive definite and (b) we achieve a 5% reduction in the total number of iterations.

6. Conclusions

We have presented an active-set algorithm for solving bound-constrained convex quadratic problems, leveraging an approach that dynamically updates both the primal and dual variables at each iteration. The algorithm efficiently determines the active set, allowing for systematic modifications that guide the solution toward feasibility and optimality while maintaining computational efficiency. This approach ensures robust convergence properties and significantly improves the solver's performance, particularly in handling large-scale problems where traditional quadratic programming methods may struggle.

Extensive experimental testing has demonstrated the superior performance of our approach compared to several well-established quadratic programming solvers. Across a variety of problem sizes and structures, the proposed algorithm exhibited faster execution times, improved numerical stability, and enhanced scalability, making it a compelling alternative for applications requiring bound-constrained optimization. The method performed particularly well in large-scale settings, where efficiently handling constraints is crucial for reducing computational overhead.

Additionally, a trust-region method for non-linear objective functions has emerged as a natural extension of our active-set framework. By integrating the proposed algorithm into the subproblem solver, the trust-region approach is capable of efficiently handling both unconstrained and bound-constrained optimization problems. The flexibility of this integration allows for enhanced adaptability in solving non-linear problems where traditional approaches may fail due to instability or attain a slow convergence.

Author Contributions: Conceptualization, I.E.L.; Software, K.V.; Validation, K.V. and I.E.L.; Writing—original draft, K.V.; Writing—review & editing, I.E.L.; Visualization, K.V.; Supervision, I.E.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Problems Descriptions

Appendix A.1. Random Problems

The Hessian matrices B have the form:

$$B = M^T M, \text{ with } M = D^{\frac{1}{2}} Z \tag{A1}$$

where

$$D = \text{diag}(d_1, \ldots, d_n), \text{ with } d_i = 10^{\frac{i-1}{n-1} ncond} \tag{A2}$$

where $ncond$ is a positive real number controlling the condition number of B

$$\kappa_2(B) = 10^{ncond}$$

and Z is a Householder matrix:

$$Z = I - 2\frac{vv^T}{v^T v} \tag{A3}$$

The vectors d, a, and b are created by the following procedure, which is controlled by two real numbers in $0, 1$, namely act_prob and up_low_prob. The algorithmic steps for the random problem creation are shown in Algorithm A1.

Algorithm A1 Random problem creation

for $i = 0$ to n **do**
  $a_i \leftarrow \text{rand}(-1, 0)$
  $b_i \leftarrow \text{rand}(0, 1)$
end for
for $i = 0$ to n **do**
  Obtain Rand $\xi_i \in [0, 1]$
  if $\xi_i \leq act\_prob$ **then**
    Obtain Rand $\tilde{\xi}_i \in [0, 1]$ ▷ Add i to the active set
    if $\tilde{\xi}_i \leq up\_low\_prob$ **then**
      $x_i \leftarrow b_i$ ▷ On upper bound
      $\mu_i \leftarrow \text{rand}(0, 1)$
      $\lambda_i \leftarrow 0$
    else
      $x_i \leftarrow a_i$ ▷ On lower bound
      $\mu_i \leftarrow 0$
      $\lambda_i \leftarrow \text{rand}(0, 1)$
    end if
  else
    $x_i \leftarrow (a_i + b_i)/2$ ▷ i in the interior
    $\mu_i \leftarrow 0$
    $\lambda_i \leftarrow 0$
  end if
end for
Calculate $d \leftarrow -Bx + \lambda - \mu$ ▷ From Equation (6)

We have created three classes of random problems:

1. Problems for which the solution has approximately 50% of the variables on the bounds, with equal probability to be either on the lower or on the upper bound ($act\_prob = 0.5$, $up\_low\_prob = 0.5$).
2. Problems for which the solution has approximately 90% of the variables on the bounds, with equal probability to be on either the lower or on the upper bound ($act\_prob = 0.9$, $up\_low\_prob = 0.5$).

3. Problems for which the solution has approximately 10% of the variables on the bounds, with equal probability to be either on the lower or on the upper bound ($act\_prob = 0.1$, $up\_low\_prob = 0.5$).

Appendix A.2. Circus Tent Problem

The circus-tent problem is a well-known example in optimization, taken from Matlab's optimization package, demonstrating large-scale quadratic programming (QP) with simple bounds. In this problem, the objective is to determine the equilibrium shape of an elastic tent supported by five poles over a square lot, while minimizing the system's potential energy under constraints imposed by the poles and the ground. The problem is formulated as a convex quadratic optimization task, where the quadratic objective function represents the elastic properties of the tent material, and the constraints enforce lower bounds set by the support poles and the ground surface [44].

Beyond this specific example, similar structural optimization problems can be posed as convex quadratic programming formulations in various topics of engineering and applied mathematics. For instance, in elastic membrane modeling, finding the equilibrium shape of a membrane under constraints involves minimizing the Dirichlet energy, which represents the elastic potential energy and leads to a quadratic objective function with bound constraints [45–47]. This formulation is particularly relevant in material science and computational physics, where deformable surfaces need to be optimized under fixed constraints.

Another key application is structural design optimization, where engineers optimize material distribution in load-bearing structures to achieve maximum efficiency while adhering to displacement and stress constraints. This optimization framework is used in architectural engineering, aerospace, and mechanical design, where lightweight yet stable structures are crucial [48]. Additionally, mechanical equilibrium analysis often involves minimizing elastic potential energy in mechanical systems with deformable components, leading to convex QP formulations [24].

These examples illustrate the broad applicability of convex quadratic programming with box constraints in structural optimization, where the goal is to determine equilibrium configurations that minimize energy functions while satisfying physical constraints.

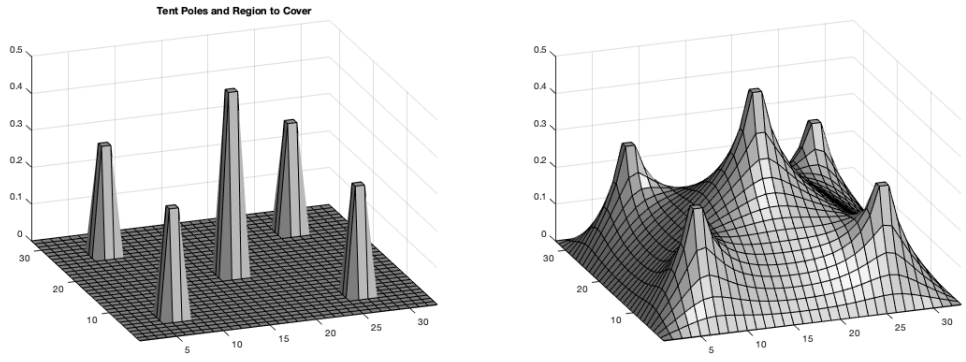


Figure A1. Circus-tent problem.

As we can see on the left side of Figure A1, the problem has only lower bounds imposed by the five poles and the ground. The surface formed by the elastic tent is determined by solving the bound constrained optimization problem:

$$\min_x f(x) = \frac{1}{2}x^T Hx + x^T c \quad \text{subject to: } l \leq x \tag{A4}$$

where $f(x)$ corresponds to the energy function and H is a 5-point finite difference Laplacian over a square grid.

Appendix A.3. Biharmonic Equation Problem

The mathematical formulation follows standard texts in elasticity theory and variational methods [49–53].

Let $\Omega \subset \mathbb{R}^2$ be a rectangular domain, and let $u : \Omega \to \mathbb{R}$ represent the vertical displacement of the membrane. The governing equation for the biharmonic problem is given by:

$$\Delta^2 u = f \quad \text{in } \Omega, \tag{A5}$$

where Δ^2 is the biharmonic operator, defined as $\Delta^2 = \Delta(\Delta u)$, and f is the external vertical force applied to the membrane [53,54]. The boundary conditions for a clamped membrane are:

$$u = 0, \quad \frac{\partial u}{\partial n} = 0 \quad \text{on } \partial\Omega. \tag{A6}$$

This ensures that the membrane is fixed along the boundary and cannot rotate. Additionally, the membrane is constrained to remain below a given obstacle function $g(x, y)$, which introduces a bound constraint:

$$u \leq g \quad \text{in } \Omega. \tag{A7}$$

Multiplying the biharmonic equation by a test function v and integrating over Ω gives the weak form:

$$\int_\Omega (\Delta u)(\Delta v)\, dx\, dy = \int_\Omega f v\, dx\, dy, \quad \forall v \in H_0^2(\Omega), \tag{A8}$$

where $H_0^2(\Omega)$ is the Sobolev space of functions with square-integrable second derivatives, satisfying the clamped boundary conditions [51]. The variational inequality formulation, incorporating the bound constraint $u \leq g$, is given by:

$$\int_\Omega (\Delta u)(\Delta(v-u))\, dx\, dy \geq \int_\Omega f(v-u)\, dx\, dy, \quad \forall v \in H_0^2(\Omega), v \leq g. \tag{A9}$$

This ensures that the solution u remains feasible under the obstacle constraint [55]. Using finite-element discretization, we approximate u in a finite-dimensional subspace $V_h \subset H_0^2(\Omega)$, where V_h is spanned by basis functions $\{\phi_i\}$. We then write:

$$u_h = \sum_{i=1}^{N} u_i \phi_i. \tag{A10}$$

The discretized bilinear form associated with the biharmonic operator leads to a symmetric positive semidefinite stiffness matrix K, giving the system:

$$Ku = F, \tag{A11}$$

where $u = (u_1, \ldots, u_N)^T$ is the vector of nodal values and F is the discrete load vector. Enforcing the bound constraint $u \leq g$ at each node, we obtain the quadratic programming problem:

$$\min_{u \in \mathbb{R}^N} \frac{1}{2} u^T K u - F^T u, \quad \text{subject to } u_i \leq g_i, \quad i = 1, ..., N. \tag{A12}$$

This is a convex quadratic program with bound constraints. We see an example in Figure A2 of a membrane under the influence of a vertical force. The function describing the force is given by:

$$f(x,y) = -60(1-x^2)y\,e^{-7(x-0.9)^2-4(y-0.1)^2} + 100x(1-y)\,e^{-3(x-0.2)^2-6(y-0.8)^2}$$

and

$$g(x,y) = 4 \cdot 10^5$$

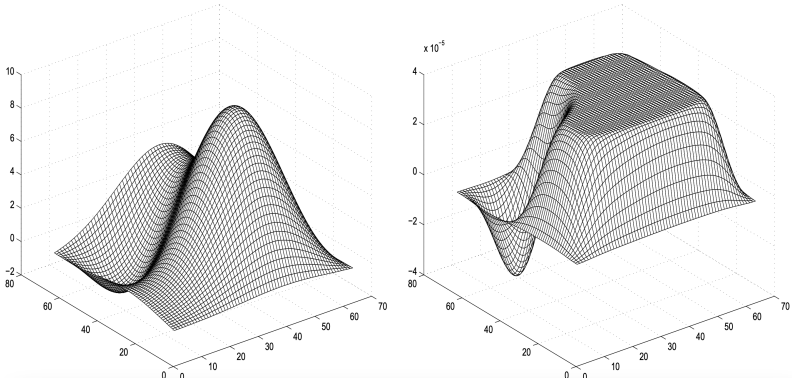


Figure A2. The force is presented on the left, and the deformation on the right.

Appendix A.4. Intensity-Modulated Radiation Therapy

The dose distribution is expressed as a linear combination of fluence elements, allowing the dose calculation to be formulated in terms of a matrix–vector representation form [56] as

$$\mathbf{d} = H\mathbf{f}, \tag{A13}$$

where $\mathbf{d}$ represents the vector of dose distributions, indicating the dose for every voxel in the patient, H refers to the dose deposition matrix, combining the distribution vectors of all beamlets, and $\mathbf{f}$ denotes the fluence vector containing the beamlet weights. In this study, the computation algorithm for H follows the method described in [57], employing a scatter radius of 3 cm. The quadratic objective function applied consists of two terms:

$$s(\mathbf{f}) = \sum_v \xi_v (H\mathbf{f} - \mathbf{d}_v^p)^T \tilde{\eta}_v (H\mathbf{f} - \mathbf{d}_v^p) + \kappa (M\mathbf{f})^T (M\mathbf{f}). \tag{A14}$$

The first term represents a commonly used quadratic dose objective that has been adapted to include voxel-specific importance factors. Here, $H\mathbf{f}$ represents the dose distribution resulting from the fluence vector $\mathbf{f}$, while $\mathbf{d}_v^p$ stands for the dose objective associated with voxels within the volume v. Each volume v is assigned an overall importance factor ξ_v along with a set of voxel-specific importance factors denoted by $\tilde{\eta}_v$. The tilde denotes the diagonal matrix form of the coefficient vector η_v. This method considers a coefficient vector whose dimension equals the total number of patient voxels, but only some coefficients are non-zero, which is determined by the implementation of voxel-specific importance factors, with a maximum number of non-zero coefficients in η_v being equal to the number of voxels in volume v.

The second term in Equation (A14) is the smoothing term, regulated by a smoothing factor κ. This term encourages the fluence $\mathbf{f}$ to be smooth. Inspired by [58], the second derivative of the fluence was used as an indicator for smoothness. If the second derivative equals zero, the fluence is linear (linearly increasing or decreasing, like a wedge or constant).

For a two-dimensional fluence, the Laplacian of the fluence $\mathbf{f}$ can be discretized using standard difference formulae for a fluence element $f_{i,j}$. With resolutions h and k of the fluence in the x- and y-direction, respectively, we have

$$\Delta f|_{(ih,jk)} = \frac{k^2(f_{i-1,j} + f_{i+1,j}) - 2(h^2 + k^2)f_{i,j} + h^2(f_{i,j-1} + f_{i,j+1})}{h^2k^2} \tag{A15}$$

The ideal case for a smooth fluence is when $\Delta f = 0$. We choose to keep the denominator h^2k^2 so the smoothing factor κ is independent of the fluence grid size. The discretization can be written in a matrix M, such that $\Delta\mathbf{f} = M\mathbf{f}$.

Equation (A14) can be written in canonical form

$$s(\mathbf{f}) = \frac{1}{2}\mathbf{f}^T A\mathbf{f} + \mathbf{f}^T\mathbf{b} + c, \tag{A16}$$

where

$$A = H^T QH + \kappa S, \qquad \mathbf{b} = H^T\mathbf{q}, \qquad c = \sum_v \xi_v (\mathbf{d}_v^p)^T \tilde{\eta}_v \mathbf{d}_v^p, \tag{A17}$$

$$Q = 2\sum_v \xi_v \tilde{\eta}_v, \qquad S = 2M^T M, \qquad \mathbf{q} = -2\sum_v \xi_v \tilde{\eta}_v \mathbf{d}_v^p. \tag{A18}$$

The scalar c in Equation (A16) can be neglected for minimization of $s(\mathbf{f})$. Matrix A is symmetric and positive definite.

Appendix A.5. Support Vector Classification

In this problem case, we are going to deal with the two-dimensional classification problem, to separate two classes using a hyperplane $f(x) = w^T x + b$, which is determined from available examples:

$$D = \{(x^1, y^1), (x^2, y^2), \ldots (x^l, y^l)\},\ x \in R^n,\ y \in \{-1, 1\}$$

Furthermore, it is desirable to produce a classifier that will work well on unseen examples, i.e., it will generalize well. Consider the example in Figure A3. There are many possible linear classifiers that can separate the data, but there is only one that maximizes the distance to the nearest data point of each class. This classifier is termed the optimal separating hyperplane and intuitively, one would expect that generalizes optimally.

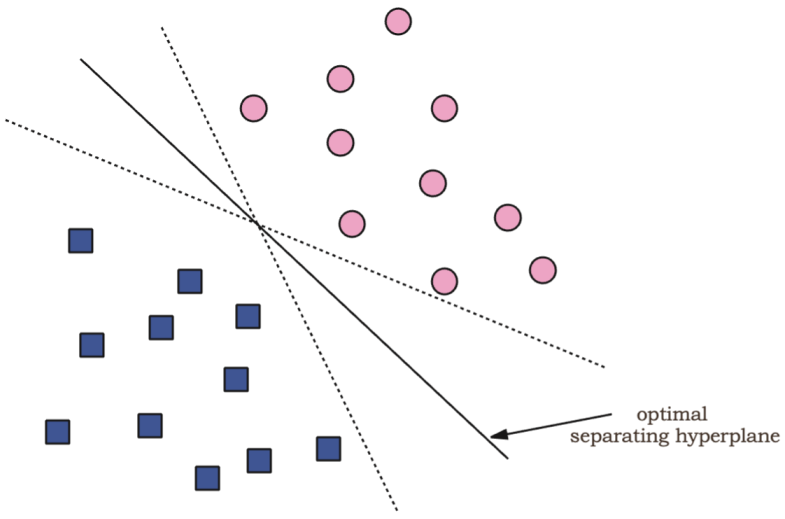


Figure A3. Maximum distance classifier.

The formulation of the maximum distance linear classifier (if we omit the constant term b of the hyperplane equation (Also known as explicit bias)) is a convex quadratic problem with simple bounds on the variables [14,59–61]. The resulting problem has the form:

$$\min_{a} \frac{1}{2} a^T Q a - a^T e \quad \text{subject to: } 0 \leq a_i \leq C \tag{A19}$$

where $e \in R^l$ and with $e_i = 1$, $Q_{ij} = y^i y^j K(x^i, x^j)$ and $K(x, y)$ is the kernel function performing the non-linear mapping into the feature space. The parameters $a \in R^l$ are Lagrange multipliers of an original quadratic problem, that define the separating hyperplane using the relation:

$$w^{*T} x = \sum_{i=1}^{l} a_i^* y^i K(x^i, x) \tag{A20}$$

Hence, the separating surface is given by:

$$f(x) = \text{sgn}(w^{*T} x) \tag{A21}$$

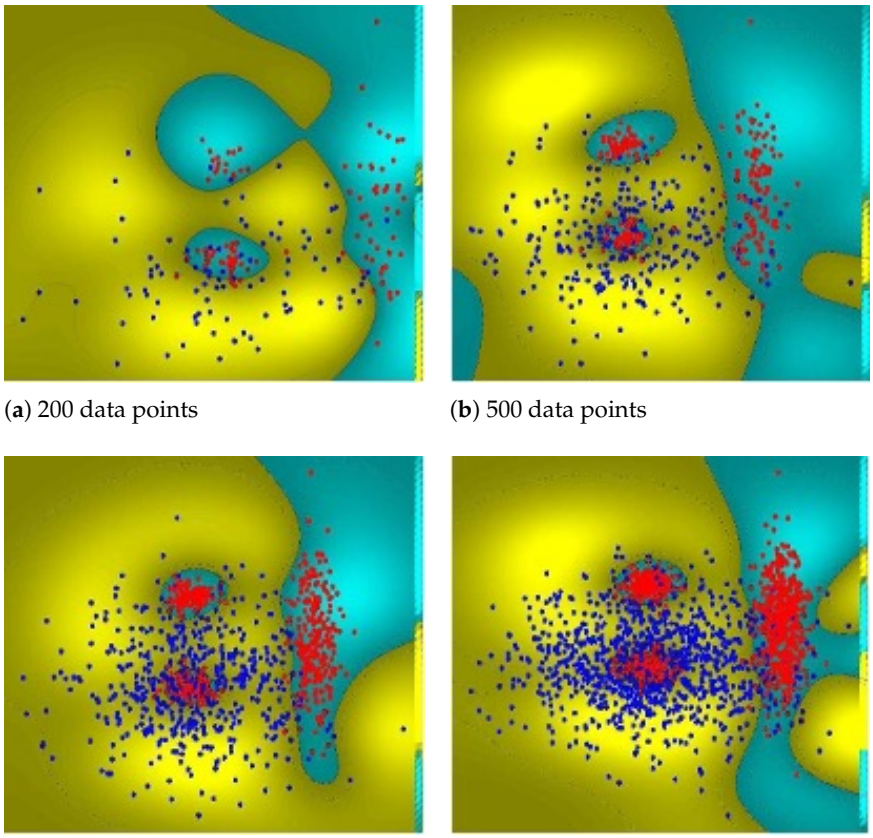
In our study, we used the two dimensional CLOUDS dataset [35], involving two distinct classes. We formulated the problem outlined in Equation (A20) by employing an RBF Kernel function defined as $K(x, y) = \exp\left(\frac{-||x-y||^2}{2C^2}\right)$, with C set to 100. The methodology for our experiments included the following steps:

1. We first extracted l examples from the dataset to create the training set, leaving the remaining (5000-l) examples for the test set.
2. Next, we constructed the matrix Q corresponding to the problem in Equation (A20).
3. Each solver was then applied to generate the separating surfaces and determine test-set errors

Notably, due to a high condition number in matrix Q, the problem became ill-conditioned, which we mitigated by adding a small positive value to the main diagonal of Q. The classification surfaces achieved for $l = 200$, 500, 1000, and 2000 training examples from the CLOUDS dataset are depicted in Figure A4.

The classification surfaces of the 2D Support Vector Machine (SVM) shown in the images evolve as the number of training data points increases. In (a) 200 data points, the decision boundary is highly irregular and appears to overfit the sparse dataset, struggling to generalize well across the feature space. There are regions with disconnected decision surfaces, indicating a lack of sufficient training samples to capture the underlying data distribution effectively.

As the number of points increases in (b) 500 data points, the decision boundary becomes more refined, though it still exhibits some irregularities, particularly in regions with complex data distributions. The increased density of support vectors (marked points along the boundary) suggests that the classifier is still adjusting to local variations. By (c) 1000 data points, the classification surface becomes smoother, demonstrating improved generalization. The previously disconnected boundary regions start forming a more coherent separation, reducing excessive curvature in low-density areas. Finally, in (d) 2000 data points, the decision surface stabilizes, capturing the overall structure of the dataset more effectively. The regions corresponding to different classes are now more clearly separated, and the classifier exhibits better robustness against noise and outliers.



(**a**) 200 data points (**b**) 500 data points

(**c**) 1000 data points (**d**) 2000 data points

Figure A4. Four instances of classification problems used in this study.

Appendix B. The Code

We present the Matlab version of the proposed quadratic programming code BoxCQP.

Listing A1. Matlab implementation.

```
function [x, iter] = boxcqp(G, d, xl, xu)
%
% Description:
%   Solves the convex quadratic minimization problem
%
%           min 1/2 x' G x + x' d
%           s.t.  xl <= x <= xu
%
% Input:
%   G : (nxn) Hessian matrix (positive definite)
%   d : (nx1) linear term
%   xl: (nx1) lower bound
%   xu: (nx1) upper bound
%
% Output:
%   x   :  (nx1) Solution
%   iter:   Number of iterations
%
    n = length(d);
    i = 1:n;
    %
    % Inititalize iteration counter
    iter = 1;
    %
    % Calculate initial point (Newton point)
```

```
    l(i) = 0;
    m(i) = 0;
    l= l';
    m = m';
    x = -G \ d;
    %
    % Check for feasibility of the Newton point
    if ( isfeas(x, xu, xl)==0)
       disp('Newton point solution: ')
       disp(x);
       return;
    end
    %
    % Loop until convergence to KKT point
    while 1>0,
        %
        % 1. Step: Define L, U, S sets
        Lset = find((x<xl | (x==xl & l>=0))>0);
        Uset = find((x>xu | (x==xu & m>=0))>0);
        Sset = find(((x>xl & x<xu)|(x==xl & l<0)|(x==xu & m<0) )>0);

        %
        % 2. Step: Projections
        x(Lset) = xl(Lset);
        m(Lset) = 0;

        x(Uset) = xu(Uset);
        l(Uset) = 0;

        m(Sset) = 0;
        l(Sset) = 0;

        %
        % 3. Step: Form reduced Ax = b!!!
        if (~isempty(Sset))
            b(Sset) = 0;
            b = b';
            if (~isempty(Lset))
               b(Sset) = b(Sset) - G(Sset,Lset)*xl(Lset);
            end
            if (~isempty(Uset))
               b(Sset) = b(Sset) - G(Sset,Uset)*xu(Uset);
            end

            b(Sset) = b(Sset) - d(Sset);

            A = G(Sset, Sset);
        %
        % 3.1 Calculate x in S
            x(Sset) = A\b(Sset);
            clear('A', 'b');

        end
        %
        % 3.2 Calculate lamda in L
        if (~isempty(Lset))
           l(Lset) = G(Lset, i) * x(i) + d(Lset) ;
        end
        %
        % 3.2 Calculate mu in U
        if (~isempty(Uset))
           m(Uset) = -G(Uset, i) * x(i) - d(Uset);
        end
```

```
89
90          % 4. Step: Check if KKT conditions are satisfied
91          iter = iter + 1;
92          f = quadratic(G, d, proj(x, xu, xl));
93          fprintf(1, 'Iter: %i, F = %f \n', iter, f);
94          if (isempty(Sset) | (x(Sset)>xl(Sset) & x(Sset)<xu(Sset)))
95              if  ((~isempty(Uset) & m(Uset)>=0 | isempty(Uset)))
96                  if ((~isempty(Lset) & l(Lset)>=0 | isempty(Lset)))
97                          disp('Point reached: ')
98                         return;
99                  end
100             end
101         end
102         %
103         % Extra check: Do not iterate more than 20 times
104         %              the problem dimension
105         if iter > n*20
106             return;
107         end
108     end
109 end
110
111 %
112 % Check if x is feasible (i.e  inside the box defined by xl, xu)
113 function res = isfeas(x, xu, xl)
114     n = length(x);
115     for i=1:n
116         if (x(i)>xu(i) | x(i)<xl(i))
117             res = 1;
118                 return;
119         end
120     end
121     res = 0;
122 end
123
124 %
125 % Projects x on the box defined by xl, xu
126 function y = proj(x, xu, xl)
127     n = length(x);
128     y = x;
129     for i=1:n
130        if (x(i)>=xu(i))
131           y(i) = xu(i);
132        elseif (x(i)<=xl(i))
133           y(i) = xl(i);
134        end
135     end
136 end
137
138 function f = quadratic(B, d, x)
139     f = 1/2 * (x' * B * x) + x'* d;
140 end
```

We present the Python version of the proposed quadratic programming code BoxCQP.

Listing A2. Python implementation.

```
1  import numpy as np
2
3  def boxcqp(G, d, xl, xu, max_iter=30, tol=1e-6):
4      """
5      Solves the convex quadratic minimization problem:
6          min 1/2 x^T G x + d^T x
7          subject to xl <= x <= xu
```

```
    :param G: (n, n) Hessian matrix (positive definite)
    :param d: (n, ) linear term
    :param xl: (n, ) lower bound
    :param xu: (n, ) upper bound
    :param max_iter: Maximum iterations
    :param tol: Tolerance for convergence
    :return: x (optimal solution), iter (number of iterations)
    """
    n = len(d)
    x = -np.linalg.solve(G, d)  # Newton step

    # Check feasibility
    if isfeas(x, xl, xu):
        #print("Newton point solution:")
        return x, 1

    iter_count = 1
    while iter_count < max_iter:
        # Define L, U, S sets
        Lset = np.where((x < xl) | ((x == xl) & (G @ x + d >= 0)))[0]
        Uset = np.where((x > xu) | ((x == xu) & (G @ x + d <= 0)))[0]
        Sset = np.setdiff1d(np.arange(n), np.concatenate((Lset, Uset)))

        # Projections
        x[Lset] = xl[Lset]
        x[Uset] = xu[Uset]

        # Solve for Sset
        if Sset.size > 0:
            b = -d[Sset]
            if Lset.size > 0:
                b -= G[np.ix_(Sset, Lset)] @ xl[Lset]
            if Uset.size > 0:
                b -= G[np.ix_(Sset, Uset)] @ xu[Uset]

            Alocal = G[np.ix_(Sset, Sset)]
            x[Sset] = np.linalg.solve(Alocal, b)

        # Compute Lagrange multipliers
        l = np.zeros(n)
        m = np.zeros(n)
        if Lset.size > 0:
            l[Lset] = G[np.ix_(Lset, np.arange(n))] @ x + d[Lset]
        if Uset.size > 0:
            m[Uset] = -G[np.ix_(Uset, np.arange(n))] @ x - d[Uset]

        # Check KKT conditions
        f_val = quadratic(G, d, proj(x, xl, xu))
        print(f"Iter: {iter_count}, F = {f_val}")

        if (np.all(m[Uset] >= 0) and np.all(l[Lset] >= 0) and
            np.all(x[Sset] > xl[Sset]) and np.all(x[Sset] < xu[Sset])):
            #print("Point reached")
            return x, iter_count

        iter_count += 1

    return x, iter_count

def isfeas(x, xl, xu):
    """Check if x is within the box constraints."""
    return np.all(x >= xl) and np.all(x <= xu)
```

```

def proj(x, xl, xu):
    """Projects x onto the box constraints xl <= x <= xu."""
    return np.clip(x, xl, xu)
def quadratic(G, d, x):
    """Computes the quadratic objective function value."""
    return 0.5 * x.T @ G @ x + d.T @ x
```

Listing A3. Python TrustBox Implementation.

```

def isPD(B):
    """Returns true when input is positive-definite, via Cholesky"""
    try:
        _ = scipy.linalg.cholesky(B)
        return True
    except scipy.linalg.LinAlgError:
        return False

def minimize(f, gradf, hessf, x0, xl, xu, delta0=1.0, deltamin=1e-6, gtol=1e-6, maxiter=100, verbose=False, use_boxcqp=False):
    """
    Solve the nonconvex optimization problem
      min_{x} f(x) subject to xl <= x <= xu
    using Newton's method, made globally convergent with trustregion.

    Simple implementation of trust-region methods, based on Algorithm 4.1 from
      Nocedal & Wright, Numerical Optimization, 2nd edn (2006)

    :param f: objective function, f : np.ndarray -> float
    :param gradf: gradient of objective, gradf : np.ndarray -> np.ndarray
    :param hessf: Hessian of objective, hessf : np.ndarray -> np.ndarray
    :param x0: starting point of solver, np.ndarray
    :param xl: lower bounds on x0, np.ndarray
    :param xu: upper bounds on x0, np.ndarray
    :param delta0: initial trust-region radius, float
    :param deltamin: final trust-region radius, float
    :param gtol: terminate when ||gradf(x)|| <= gtol, float
    :param maxiter: terminate after maxiter iterations
    :param verbose: whether to print information at each iteration, bool
    :return: solution x, number of iterations
    """
    xk = np.maximum(np.minimum(xu, x0), xl)  # current iterate, project x0 into box
    deltak = delta0
    if verbose:
        print("{0:^10}{1:^10}{2:^15}{3:^15}".format("k", "f(xk)", "||gradf(xk)||", "xk"))
        np.set_printoptions(precision=4, suppress=False)
    k = -1

    d = len(xk)
    B = np.eye(len(xk))
    nabla = gradf(xk) # initial gradient

    while k < maxiter:
        k += 1
        # Evaluate objective
        fk = f(xk)
        gk = gradf(xk)
        Hk = hessf(xk)
        # With box constraints, we have a new criticality measure
```

```
        # See Theorem 12.1.6 of Conn, Gould & Toint, Trust-Region Methods (2000)
        # Note: if unbounded, this reduces to np.linalg.norm(gk)
        crit_measure = abs(np.dot(gk, trustregion.solve(gk, None, 1.0, sl=xl - xk, su=xu - xk)))
        if verbose:
            print("{0:^10}{1:^10.4e}{2:^15.4e}{3:^15}".format(k, fk, crit_measure, str(xk)))
        # Check termination
        if crit_measure <= gtol or deltak <= deltamin:
            break  # quit loop
        # Step calculation

        if use_boxcqp and isPD(Hk):
            if verbose:
                print ('isPD')
            ss = boxcqp(Hk, gk,
                  np.maximum(xl-xk, -deltak*np.ones(len(xk))),
                  np.minimum(xu-xk,  deltak*np.ones(len(xk))))
            sk = ss[0]
        else:
            if verbose:
                print ('not')
            sk = trustregion.solve(gk, Hk, deltak, sl=xl-xk, su=xu-xk)


        model_value = fk + np.dot(gk, sk) + 0.5 * np.dot(sk, Hk.dot(sk))  # mk(sk)
        rhok = (fk - f(xk + sk)) / (fk - model_value)
        # Update trust-region radius
        if rhok < 0.25:
            deltak = 0.25 * deltak
        elif rhok > 0.75 and abs(np.linalg.norm(sk) - deltak) < 1e-10:
            deltak = min(2 * deltak, 1e10)
        else:
            deltak = deltak
        # Update iterate
        if rhok > 0.01:
            xk = xk + sk
        else:
            xk = xk
    return xk, k
```

References

1. Sun, W.; Yuan, Y.X. *Optimization Theory and Methods: Nonlinear Programming*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2006; Volume 1.
2. Fletcher, R. *Practical Methods of Optimization*, 2nd ed.; John Wiley & Sons, Inc.: New York, NY, USA, 1987.
3. Gill, P.E.; Murray, W.; Wright, M.H. *Practical Optimization*; Academic Press: New York, NY, USA, 1981.
4. Voglis, C.; Lagaris, I.E. A Rectangular Trust-Region Approach for Unconstrained and Box-Constrained Optimization Problems. In Proceedings of the International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2004), Athens, Greece, 4–7 April 2004.
5. Hungerlander, P.; Rendl, F. A feasible active set method for strictly convex quadratic problems with simple bounds. *SIAM J. Optim.* **2015**, *25*, 1633–1659. [CrossRef]
6. Schwan, R.; Jiang, Y.; Kuhn, D.; Jones, C.N. PIQP: A Proximal Interior-Point Quadratic Programming Solver. In Proceedings of the 2023 62nd IEEE Conference on Decision and Control (CDC), Singapore, 13–15 December 2023; pp. 1088–1093. [CrossRef]
7. Bambade, A.; Schramm, F.; El Kazdadi, S.; Caron, S.; Taylor, A.; Carpentier, J. Proxqp: An Efficient and Versatile Quadratic Programming Solver for Real-Time Robotics Applications and Beyond. 2023. Available online: https://inria.hal.science/hal-04198663/ (accessed on 1 February 2025).
8. Yue, H.; Shen, P. Bi-affine scaling iterative method for convex quadratic programming with bound constraints. *Math. Comput. Simul.* **2024**, *226*, 373–382. [CrossRef]

9. Ranasinghe, L.; Disanayake, A.; Cooray, T. *Portfolio Optimization Using Quadratic Programming*; University of Moratuwa, Institutional Repository: Moratuwa, Sri Lanka, 2020.
10. Markowitz, H. Portfolio Selection. *J. Financ.* **1952**, *7*, 77–91.
11. Kontosakos, V.E. Fast Quadratic Programming for Mean-Variance Portfolio Optimisation. In *SN Operations Research Forum*; Springer International Publishing: Cham, Switzerland, 2020; Volume 1, p. 25.
12. Bodnar, T.; Parolya, N.; Schmid, W. On the Equivalence of Quadratic Optimization Problems Commonly Used in Portfolio Theory. *arXiv* **2012**, arXiv:1207.1029. [CrossRef]
13. Osuna, E.; Freund, R.; Girosi, F. An Improved Training Algorithm for Support Vector Machines. In Proceedings of the IEEE Workshop on Neural Networks for Signal Processing, Amelia Island, FL, USA, 24–26 September 1997; pp. 276–285.
14. Platt, J.C. Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. In *Advances in Kernel Methods—Support Vector Learning*; MIT Press: Cambridge, UK, 1998; pp. 185–208.
15. Gunn, S. *Support Vector Machines for Classification and Regression*; Technical Report, ISIS Technical Report; Image Speech & Intelligent Systems Group, University of Southampton: Southampton, UK, 1997.
16. Osuna, E.; Freund, R.; Girosi, F. *Support Vector Machines: Training and Applications*; Technical Report; Massachusetts Institute of Technology: Cambridge, MA, USA, 1997.
17. Bro, R.; De Jong, S. Fast Non-Negativity-Constrained Least Squares Regression. *J. Chemom.* **1997**, *11*, 393–401. [CrossRef]
18. Lawson, C.L.; Hanson, R.J. *Solving Least Squares Problems*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1995.
19. Rawlings, J.B.; Mayne, D.Q.; Diehl, M. *Model Predictive Control: Theory, Computation, and Design*; Nob Hill Publishing: San Francisco, CA, USA, 2017.
20. Diehl, M.; Bock, H.G.; Schloder, J. Real-Time Optimization for Large-Scale Nonlinear Processes. *SIAM J. Optim.* **2005**, *15*, 837–860.
21. Breedveld, S.; Storchi, P.R.M.; Keijzer, M.; Heijmen, B.J.M. Fast, multiple optimizations of quadratic dose objective functions in IMRT. *Phys. Med. Biol.* **2006**, *51*, 3569. [CrossRef]
22. Conn, A.R.; Gould, N.I.M.; Toint, P.L. *LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*; Springer Series in Computational Mathematics; Springer: Berlin/Heidelberg, Germany, 1992; Volume 17.
23. Bertsekas, D.P. *Nonlinear Programming*; Athena Scientific: Nashua, NH, USA, 1996.
24. Boyd, S.P.; Vandenberghe, L. *Convex Optimization*; Cambridge University Press: Cambridge, UK, 2004.
25. Lu, H.; Yang, J. A practical and optimal first-order method for large-scale convex quadratic programming. *arXiv* **2023**, arXiv:2311.07710.
26. Voglis, C.; Lagaris, I.E. BOXCQP: An algorithm for bound constrained convex quadratic problems. In Proceedings of the 1st International Conference: From Scientific Computing to Computational Engineering, IC-SCCE, Athens, Greece, 8–10 September 2004.
27. Kunisch, K.; Rendl, F. An Infeasible Active Set Method For Quadratic Problems With Simple Bounds. *SIAM J. Optim.* **2003**, *14*, 35–52. [CrossRef]
28. Nocedal, J.; Wright, S.J. Conjugate gradient methods. In *Numerical Optimization*; Springer: New York, NY, USA, 2006; pp. 101–134.
29. Hager, W.W.; Zhang, H. A survey of nonlinear conjugate gradient methods. *Pac. J. Optim.* **2006**, *2*, 35–58.
30. Madsen, K.; Nielsen, H.B.; Pinar, M.C. Bound Constrained Quadratic Programming via Piecewise Quadratic Functions. *Math. Program.* **1999**, *85*, 135–156. [CrossRef]
31. Schittkowski, K. *QLD: A FORTRAN Code for Quadratic Programming, Users Guide*; Technical Report; Mathematisches Institut, Universität Bayreuth: Bayreuth, Germany, 1986.
32. More', J.J.; Toraldo, G. On the solution of quadratic programming problems with bound constraints. *SIAM J. Optim.* **1991**, *1*, 93–113. [CrossRef]
33. Friedlander, A.; Martinez, M. On the maximization of a concave quadratic function with box constraints. *SIAM J. Optim.* **1994**, *4*, 177–192. [CrossRef]
34. Breedveld, S. (Department of Radiotherapy Erasmus MC Cancer Institute (Erasmus University Medical Center), Rotterdam, The Netherlands). Private communication, 2015.
35. Blake, C.L.; Merz, C.J. UCI Repository of Machine Learning Databases. 1998. Available online: http://www.ics.uci.edu/~mlearn/MLRepository.html (accessed on 1 February 2025).
36. Coleman, T.F.; Li, Y. An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds. *SIAM J. Optim.* **1996**, *6*, 418–445. [CrossRef]
37. Stark, P.B.; Parker, R.L. Bounded-variable least-squares: An algorithm and applications. *Comput. Stat.* **1995**, *10*, 129.
38. Chang, T.J.; Meade, N.; Beasley, J.E.; Sharaiha, Y.M. Heuristics for cardinality constrained portfolio optimisation. *Comput. Oper. Res.* **2000**, *27*, 1271–1302. [CrossRef]
39. Dantas, G.; Blondel, M.; Cournapeau, D. qpsolvers: A Unified Interface for Quadratic Programming Solvers in Python. 2020. Available online: https://qpsolvers.github.io/qpsolvers (accessed on 1 February 2025).

40. Stellato, B.; Banjac, G.; Goulart, P.; Bemporad, A.; Boyd, S. OSQP: An Operator Splitting Solver for Quadratic Programs. *Math. Program. Comput.* **2020**, *12*, 637–672. [CrossRef]
41. Conn, A.R.; Gould, N.I.; Toint, P.L. *Trust Region Methods*; SIAM: Philadelphia, PA, USA, 2000.
42. Byrd, R.H.; Gilbert, J.C.; Nocedal, J. A trust region method based on interior point techniques for nonlinear programming. *Math. Program.* **2000**, *89*, 149–185. [CrossRef]
43. Dennis, J.E.; Schnabel, R.B. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; SIAM: Philadelphia, PA, USA, 1996.
44. MathWorks. Large-Scale Problem-Based Quadratic Programming. 2024. Available online: https://www.mathworks.com/help//releases/R2021a/optim/ug/large-scale-problem-based-quadratic-programming.html (accessed on 17 March 2025).
45. Kanno, Y. Exploiting Lagrange Duality for Topology Optimization with Frictionless Unilateral Contact. *arXiv* **2020**, arXiv:2011.07732.
46. Bołbotowski, K. Optimal Design of Plane Elastic Membranes Using the Convexified Föppl's Model. *Appl. Math. Optim.* **2023**, *90*, 23. [CrossRef]
47. Burtscheidt, J.; Claus, M.; Conti, S.; Rumpf, M.; Sassen, J.; Schultz, R. A Pessimistic Bilevel Stochastic Problem for Elastic Shape Optimization. *Math. Program.* **2023**, 198, 1125–1151. [CrossRef]
48. Bendsoe, M.P.; Sigmund, O. *Topology Optimization: Theory, Methods, and Applications*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2013.
49. Strang, G. *Introduction to Applied Mathematics*; Wellesley-Cambridge Press: Wellesley, MA, USA, 1986.
50. Lions, J.L. *Quelques Méthodes De Résolution Des Problèmes Aux Limites*; Dunod: Malakoff, France, 1969.
51. Glowinski, R. *Numerical Methods for Nonlinear Variational Problems*; Springer: Berlin/Heidelberg, Germany, 2008.
52. Duvaut, G.; Lions, J.L. *Inequalities in Mechanics and Physics*; Springer: Berlin/Heidelberg, Germany, 1976.
53. Braess, D. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*; Cambridge University Press: Cambridge, UK, 2007.
54. Gilat, A.; Subramaniam, V. *Numerical Methods for Engineers and Scientists*; Wiley: Hoboken, NJ, USA, 2008.
55. Kinderlehrer, D.; Stampacchia, G. *An Introduction to Variational Inequalities and Their Applications*; SIAM: Philadelphia, PA, USA, 1980.
56. Cho, P.S.; Phillips, M.H. Reduction of computational dimensionality in inverse radiotherapyplanning using sparse matrix operations. *Phys. Med. Biol.* **2001**, *46*, N117. [CrossRef] [PubMed]
57. Storchi, P.; Woudstra, E. Calculation of the absorbed dose distribution due to irregularly shaped photon beams using pencil beam kernels derived from basic beam data. *Phys. Med. Biol.* **1996**, *41*, 637. [CrossRef] [PubMed]
58. Webb, S.; Convery, D.; Evans, P. Inverse planning with constraints to generate smoothed intensity-modulated beams. *Phys. Med. Biol.* **1998**, *43*, 2785. [CrossRef]
59. Boser, B.E.; Guyon, I.M.; Vapnik, V.N. A Training Algorithm for Optimal Margin Classifiers. In Proceedings of the Fifth Annual Workshop on Computational Learning Theory, Pittsburgh, PA, USA, 27–29 July 1992; pp. 144–152.
60. Cortes, C.; Vapnik, V. Support-vector networks. *Mach. Learn.* **1995**, *20*, 273–297. [CrossRef]
61. Schölkopf, B.; Smola, A.J. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*; MIT Press: Cambridge, UK, 2001.