Quadratic Programming: Algorithms and Applications

Ji Ran, Fang Youzheng, Xu Xinyue

January 11, 2025

An in-depth exploration of optimization theory and practical implementations

1. Background and Importance of Quadratic Programming (QP)

1.1 Background

Quadratic Programming (QP) is a significant branch of optimization theory, where the objective function is quadratic, and the constraints are linear. The standard form of a QP problem is:

$$\min \frac{1}{2}x^T Qx + c^T x$$

$$s.t. Ax \leq b,\ Ex = d$$

Here:

- Q is a symmetric positive definite or semi-definite matrix defining the quadratic term of the objective function.
- c is the coefficient vector of the linear term.
- A, b and E, d represent the matrices and vectors for the inequality and equality constraints, respectively.

QP is an extension of Linear Programming (LP), with a more general objective function, enabling the representation of a broader range of optimization problems.

1.2 Importance

1. **Wide Application Areas**
    - **Machine Learning:** QP is widely used in models such as Support Vector Machines (SVMs), where the goal is to maximize the classification margin, inherently formulated as a QP problem.
    - **Economics and Finance:** In portfolio optimization, the aim is to maximize returns while minimizing risks, often modeled using a quadratic objective function.
    - **Engineering and Control:** QP is applied in trajectory optimization, path planning, and optimal control of dynamic systems.
    - **Signal Processing:** QP is utilized in problems like beamforming and resource allocation.
2. **Theoretical Significance**
    - QP enriches the study of optimization theory as a nonlinear problem. It serves as a foundation for exploring more complex optimization problems such as non-convex optimization and multi-objective optimization.
    - Performance analysis of classical algorithms (e.g., interior-point methods) on QP problems often reflects their potential for tackling more complex challenges.
3. **Computational Complexity Challenges**
    - Although QP is more complex than LP, significant progress has been made in its solution techniques, with classical methods like interior-point algorithms achieving high efficiency in practice.
    - With the rise of high-dimensional data and large-scale problems, developing efficient QP solvers is critical for numerous disciplines.
4. **A Bridge Between Theory and Practice**
    - As a vital tool in mathematical optimization, QP holds both theoretical and practical significance. Researching QP problems enables direct applications of mathematical theories to real-world scenarios.

In summary, QP is a key problem in mathematical optimization, playing a crucial role in various fields and advancing the interplay between theoretical research and practical applications.

2. Analysis of several major existing algorithms and their literature

2.1 Active-set methods: qpOASES, qrqp

Take this literature as an example: https://www.semanticscholar.org/paper/qpOASES%3A-a-parametric-active-set
ca3a7e977f1ca6edc5b39483c5b1d7f7b0b7d02e?utm_source=consensus.

1. **Core Methodology**

   qpOASES is an open-source C++ software package implementing a parametric active-set algorithm, designed to efficiently solve convex quadratic programming (QP) problems. The algorithm traces optimal solutions along a homotopy path between two QP instances parameterized by $\tau \in [0, 1]$.

   - This approach efficiently handles sequences of QP problems with gradually changing parameters, especially in real-time applications like Model Predictive Control (MPC).
   - The algorithm dynamically maintains and updates an active set of constraints, leveraging the similarity between related QP problems to achieve rapid convergence through warm-starting.

2. **QP Solver Details**

   - qpOASES employs a primal-dual active-set strategy, solving a sequence of linear systems derived from the Karush-Kuhn-Tucker (KKT) conditions.
   - The solver handles both equality and inequality constraints and exploits problem structures to enhance computational performance.
   - It provides various initialization and warm-starting techniques, making it ideal for scenarios where QP problems are solved repeatedly with slight variations, such as in MPC.

3. **Benchmarks and Performance Evaluation**

   - qpOASES was evaluated using the Maros-Mészáros QP collection, a standard set of test cases for QP solvers.
   - Results demonstrate that qpOASES performs competitively with other academic and commercial QP solvers, particularly on small- and medium-scale convex problems.
   - Its efficiency in these benchmarks highlights its suitability for real-time applications requiring high computational speed.

4. **Additional Information**

   - qpOASES features an object-oriented architecture and user-friendly interfaces, facilitating integration into diverse applications.
   - It supports different types of QP problems, including those with varying Hessian and constraint matrices.
   - The solver extends its applicability to nonconvex QP problems by computing critical points.

   In summary, qpOASES is a robust and efficient tool for solving QP problems, with significant advantages in parameter-varying QP scenarios, such as Model Predictive Control (MPC).

2.2 Interior-point methods: hpipm, OOQP, CVXGEN

Take this literature as an example: https://www.semanticscholar.org/paper/Optimization-Ho/458aa276338c7805d55cb4bcfefb4f34869f9733?utm_source=consensus

1. **Core Methodology**

   (a) **Quadratic Programming Framework:**

   - HPIPM is an open-source C-coded framework supporting three QP types: dense QP, optimal control problem (OCP) QP, and tree-structured OCP QP.
   - It provides interior point method (IPM) solvers and (partial) condensing routines tailored for these QP types.

(b) **Modular Design:**
   - The framework adopts a modular design, enabling easy extension for new QP types and serving as a basis for more general algorithms, such as NLP solvers.
   - Encapsulation mechanisms in its C API ensure internal data structures are accessed through setters and getters, improving maintainability and user-friendliness.

(c) **Memory Management:**
   - HPIPM operates on externally provided memory chunks, with no internal memory allocation, making it suitable for embedded applications.
   - Interfaces are available for Matlab, Octave, Simulink, and Python.

2. **Details of the QP Solver**

(a) **Interior Point Method (IPM) Solvers:**
   - HPIPM provides a family of IPM solvers with different trade-offs between computational speed and robustness.
   - The fastest algorithm is analogous to HPMPC but with improved robustness.

(b) **KKT System Handling:**
   - The framework includes routines for factorizing and solving the Karush-Kuhn-Tucker (KKT) system, tailored to specific QP types.
   - These routines are implemented using BLASFEO, a framework for efficient linear algebra computations.

(c) **(Partial) Condensing Routines:**
   - HPIPM implements condensing and partial condensing routines, converting an OCP QP into a dense QP or an OCP QP with a shorter horizon.
   - These routines can be used standalone or coupled with other QP solvers.

3. **Benchmarks**

(a) **Performance Evaluation:**
   - Numerical experiments show HPIPM reliably solves challenging QPs and outperforms other state-of-the-art solvers in terms of speed.
   - The framework's robustness and efficiency make it suitable for real-time applications.

(b) **Comparison with Other Solvers:**
   - HPIPM demonstrates superior performance compared to existing solvers for efficiently solving QPs.
   - Its focus on speed without compromising robustness makes it a strong choice for MPC applications.

4. **Additional Information**

(a) **Applications in Model Predictive Control:**
   - HPIPM is specifically designed for MPC needs, providing efficient and reliable solutions for linear-quadratic optimal control problems.
   - Its modularity and performance make it suitable for a wide range of MPC applications, including real-time scenarios.

(b) **Integration with Other Tools:**
   - The framework's high-level language interfaces allow seamless integration into existing workflows and control systems.
   - Compatibility with BLASFEO enhances computational efficiency, especially for embedded optimization.

2.3 Alternating direction method of multipliers (ADMM): OSQP

Take this literature as an example: `https://www.semanticscholar.org/paper/Embedded-code-generation-using-the`
`93571f33c7621edf5ef98686b5d1889564a5f4e9?utm_source=consensus`

1. **Core Methodology**

   (a) **Operator Splitting and ADMM:** The OSQP solver uses operator splitting techniques combined with the Alternating Direction Method of Multipliers (ADMM).

   - Operator splitting decomposes the complex QP problem into two simpler subproblems that are solved iteratively.
   - ADMM provides a stable and efficient iterative framework suitable for sparse or large-scale problems.

   (b) **Embedded Code Generation:** The paper introduces a code generation tool that produces optimized C code tailored to parametric QP problems.

   - The code is highly modular, malloc-free, and division-free after initialization, ensuring reliability.
   - The generated code is compact and memory-efficient, making it ideal for resource-constrained embedded platforms.

   (c) **Real-Time Capabilities and Flexibility:** The generated code supports parameter changes (e.g., vectors q, l, u) while maintaining the problem's structure.

   - This flexibility is essential for real-time applications such as Model Predictive Control (MPC).

2. **QP Solver Details**

   - **Matrix Factorization and Caching:**
     - The OSQP algorithm solves a quasi-definite linear system at its core, where the coefficient matrix remains constant in most iterations.
     - Pre-computed matrix factorizations (e.g., sparse Cholesky factorization) are cached to reduce computation and enhance speed.
   - **Robustness:** OSQP is designed to detect:
     - Primal feasibility.
     - Dual feasibility.

     Numerical safeguards and specific stopping criteria improve the algorithm's stability and convergence.
   - **Numerical Precision and Efficiency:**
     - Sparse matrix operations optimize performance for embedded systems with limited computational resources.
     - Adjustable step sizes and Lagrange multiplier updates ensure high convergence rates.

3. **Benchmarks**

   (a) **Testing Environment:** The paper evaluates the generated code using various real-world and simulated scenarios, including:

   - Model Predictive Control (MPC).
   - Financial optimization problems.
   - Robotic motion control.

   (b) **Performance Comparison:**

   - Compared to the standard OSQP solver and other general-purpose QP solvers (e.g., CPLEX, GUROBI), the embedded code demonstrates:
     - Faster execution.
     - Lower memory usage.

    - Particularly on embedded hardware (e.g., microcontrollers, FPGAs), the generated code excels in real-time performance.

4. **Additional Information**
    - **Code Interface and Usage:**
        - The generated code supports multiple programming languages (e.g., C, C++, Python, Matlab) and is designed for seamless integration.
        - Users can define QP problems through high-level interfaces and automatically generate low-level optimization code.
    - **Applications:**
        - Path planning in autonomous vehicles.
        - Optimal scheduling in power systems.
        - Embedded robotic control requiring real-time responsiveness and stability.

In conclusion, the paper presents a code generation tool leveraging OSQP's flexibility and efficiency. The customized code generation method optimized for embedded systems exhibits outstanding performance in real-time, resource utilization, and robustness, providing a reliable solution for QP problems in embedded applications.

3. Implement

Improvements to deal with non-convex and infeasible case

For a quadratic programming problem, the matrix H must be positive semi-definite for the problem to be convex. Our algorithm checks if any eigenvalue is negative. If any eigenvalue is negative, the matrix is not positive semi-definite, meaning the problem might be non-convex. If the matrix H is found to be non-positive semi-definite (i.e., it has negative eigenvalues), Adjustment is made to ensure that H becomes positive semi-definite as follows:

$$\mathbf{H}' = \mathbf{H} - \mathbf{I} \cdot \lambda_{min}$$

where λ_{min} is the smallest eigenvalue of H. We add λ_{min} to the diagonal elements of H to make all eigenvalues non-negative. This small adjustment effectively ensures that the matrix H becomes positive semi-definite, thereby ensuring the convexity of the quadratic programming problem.

After obtaining the solution x from the algorithm, our algorithm performs the equality constraints check and inequality constraints check. The check is performed after each iteration and after convergence. If the solution violates any of the constraints at any point, the function prints a warning message indicating that the problem is infeasible.

4. Test

4.1 Analyzing the impact of parameter variations on IRWA

In the experiment, we generates a random feasible and convex quadratic programming problem by randomly creating the objective function coefficients and constraints that satisfy the necessary conditions for convexity and feasibility. H is generated by first creating a random matrix Q with shape (n, n) and then computing $H = Q^T Q + I$, where I is the identity matrix. This ensures that H is positive definite.

In this part, we choose some parameters like γ and η to analyze their impact on iteration numbers and the objective function values.

Firstly, we intend to investigate the effect of η on the number of iterations and the function values. We control all variables except η, allowing η to vary from 0.5 to 0.95, and plot the curves of η versus iterations and η versus function values.

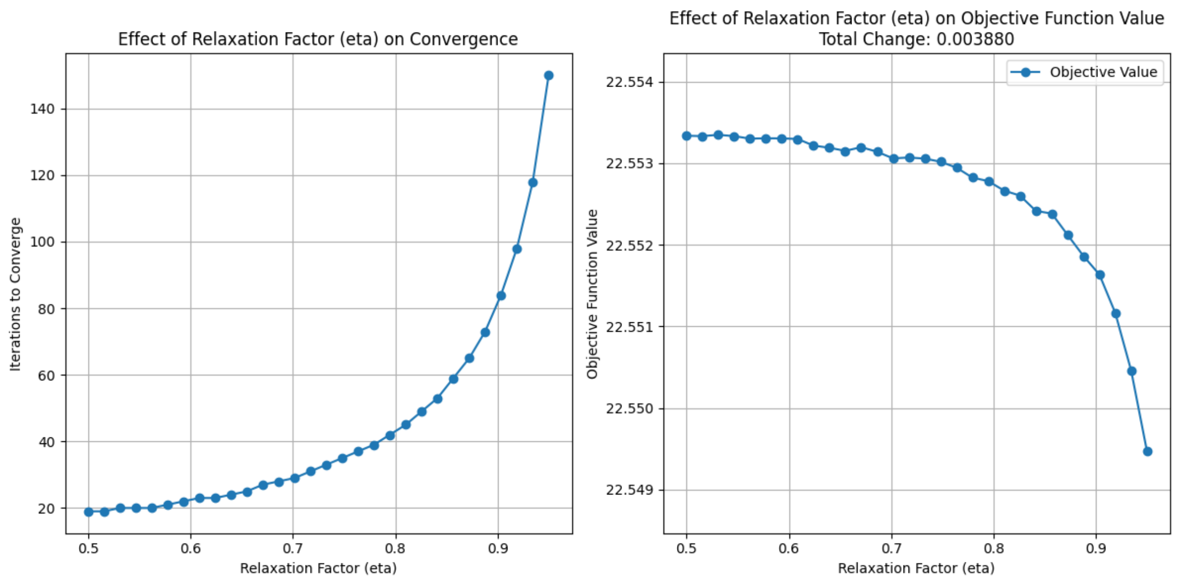

Figure 1: Effect of relaxation factor on performance of IRWA.

From the figure, it can be seen that as η increases, the number of iterations of the algorithm rises significantly, and the function values gradually decrease, though the decrease is not substantial.

Next, we examined the effect of γ on the number of iterations and the function values. Similar to the study of η, this time we controlled all variables except γ, allowing γ to vary from 0.01 to 10, and plotted the curves of γ versus iterations and γ versus function values.

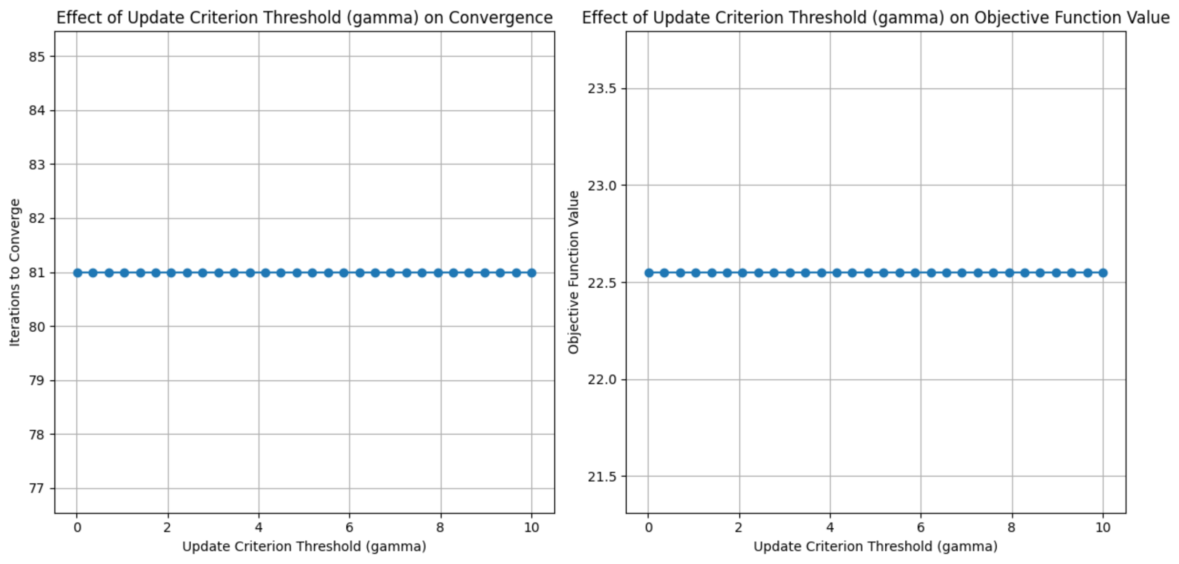

Figure 2: Effect of update criterion threshold on performance of IRWA.

From the figure, it can be seen that regardless of how γ varies between 0.01 and 10, neither the number of iterations nor the function values of the algorithm change. This indicates that the value of γ has little overall impact on the algorithm.

Next, we examined the effects of M_1 and M_2 (i.e., the equality constraint penalty coefficient and the inequality constraint penalty coefficient) on the number of iterations and the function values of the algorithm.

Firstly, we controlled M_2 and kept the other parameters unchanged, allowing M_1 to vary from 1 to 200. Then, we plotted the curves of M_1 versus the number of iterations and M_1 versus the function values.

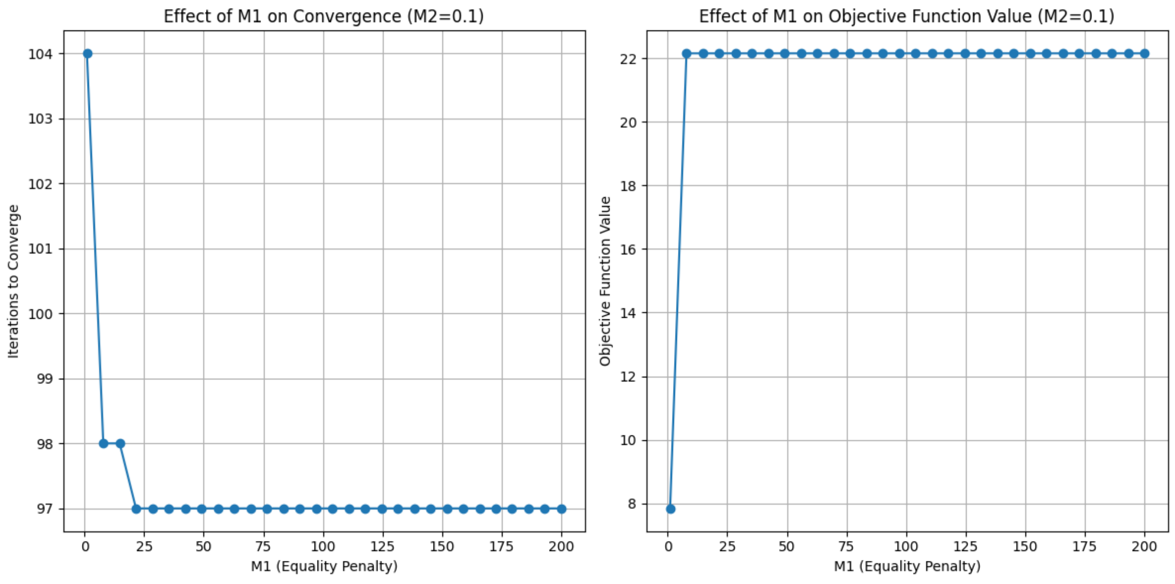

Figure 3: Effect of M1 on performance of IRWA.

From the figure, it can be seen that except when the value of M_1 is extremely small (i.e., 1), M_1 has almost no effect on the number of iterations and the function values of the algorithm. Secondly, we controlled M_1 and kept the other parameters unchanged, allowing M_2 to vary from 0.1 to 50. Then, we plotted the curves of M_2 versus the number of iterations and M_2 versus the function values.

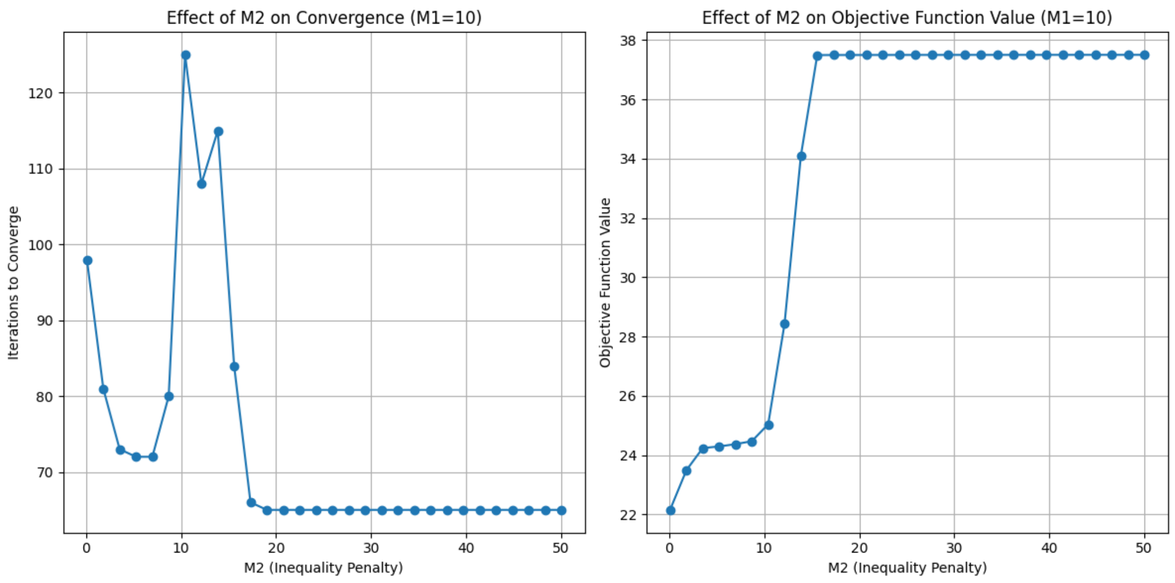

Figure 4: Effect of M2 on performance of IRWA.

From the figure, it can be seen that as M_2 gradually increases, the number of iterations of the algorithm exhibits significant fluctuations, but overall still shows a decreasing trend. This indicates that increasing M_2 can accelerate the convergence speed of the algorithm. The function values also increase with the rise of M_2 and eventually stabilize.

Finally, we selected several combinations of M_1 and M_2 and, while keeping the other variables unchanged, studied the convergence of the algorithm by varying these combinations. We used the values of M_1 and M_2 as the horizontal and vertical axes, respectively, and represented the number of iterations with the color of the points. The color gradient from light to dark indicates that the algorithm requires more iterations to converge.

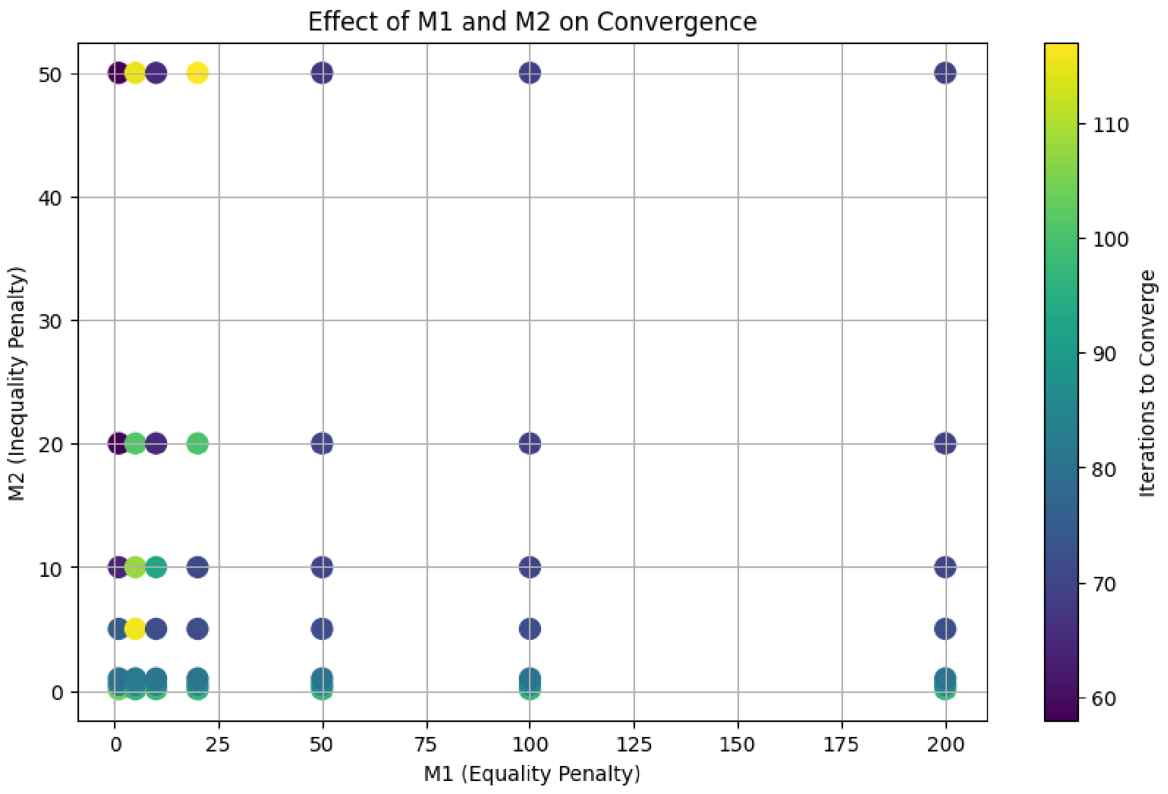

Figure 5: Effect of M2 on convergence of IRWA.

4.2 Analyzing the impact of parameter variations on ADAL

This experiment adopts the same problem generation method as the experiment on IRWA in 3.1.

Firstly, we investigate the effect of μ on the number of iterations and the function values. We control all variables except μ, allowing μ to vary from 1 to 100, and plot the curves of μ versus iterations and function values. From the figure, it can be seen that as μ increases, both the number of iterations of the algorithm and the function values rises significantly.

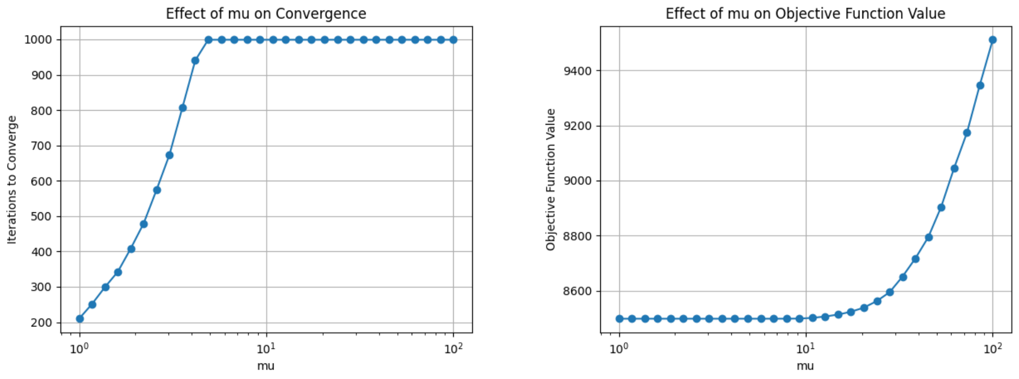

Figure 6: Effect of mu on convergence of ADAL.

Next, we examined the effect of σ on the log of the number of iterations and the function values. We controlled all variables except σ. From the figure, it can be seen that as σ increases, both the number of iterations of the algorithm and the function values decreases significantly.

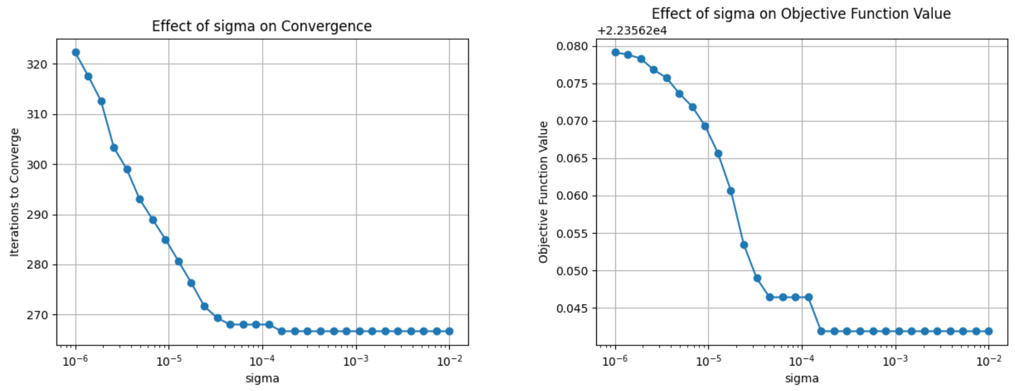

Figure 7: Effect of sigma on convergence of ALAL.

Then we examined the effect of σ'' on the logarithm of the number of iterations and the function values. As σ'' increases, the number of iterations of the algorithm decreases linearly. But the function values increases as σ'' rises.

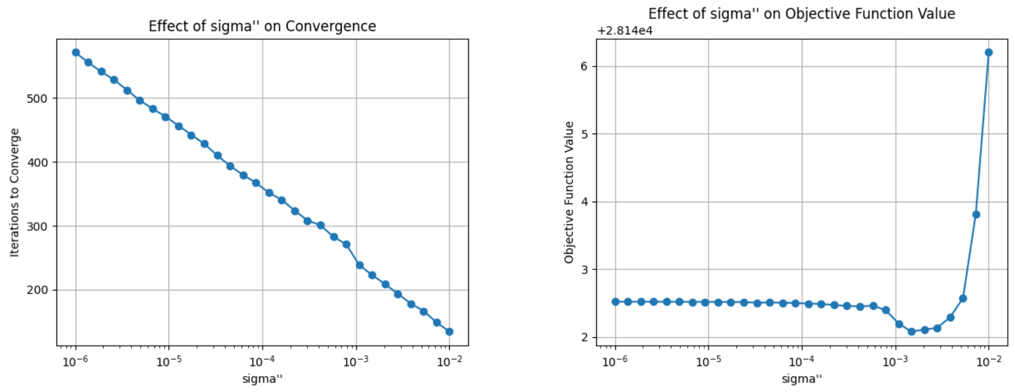

Figure 8: Effect of sigma" on convergence of ALAL.

Similar to the experiments on IRWA, we also examined the effects of M_1 and M_2 on the number of iterations and the function values of ADAL.

Firstly, we controlled M_2 and kept the other parameters unchanged and change $M1$. The figure plot the curves of M_1 versus the number of iterations and M_1 versus the function values. It can be seen that as M_1 is large enough, the number of iterations slightly decreases and the function values of the algorithm slightly rises.

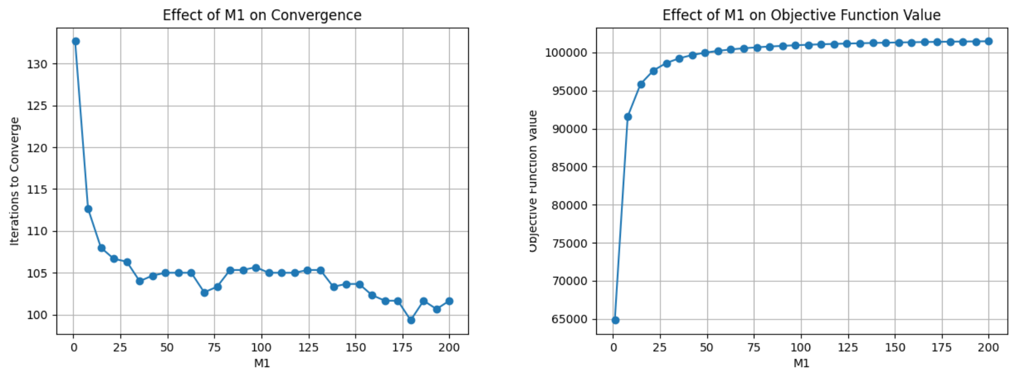

Figure 9: Effect of M1 on convergence of ALAL.

Secondly, we controlled M_1 and kept the other parameters unchanged, allowing M_2 to vary. Then, we plotted the curves of M_2 versus the number of iterations and M_2 versus the function values. Similarly to $M1$, as M_2 is large enough, the number of iterations slightly decreases and the function values of the algorithm slightly rises.

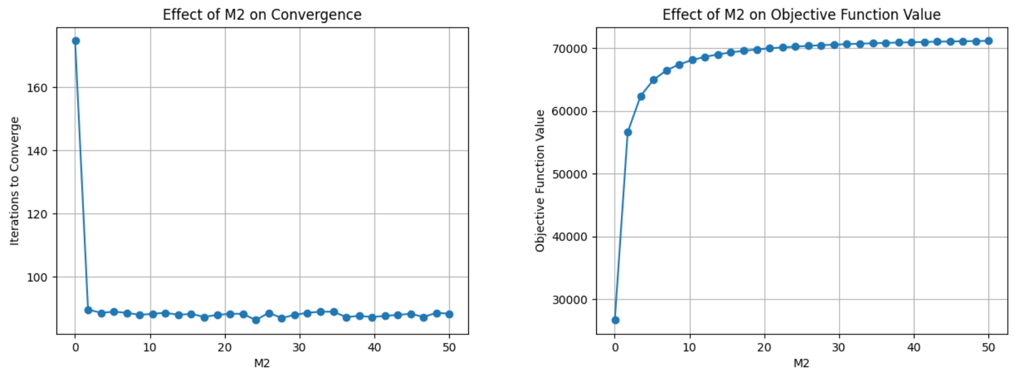

Figure 10: Effect of M2 on convergence of ALAL.

4.3 Analyzing the impact of tolerance

In this experiment, we randomly generated 500 instances. For each, we generated $A \in R^{12\times 20}$. The matrix A is generated for both equality and inequality constraints. It has $m_{eq} + m_{ineq}$ rows (for the number of constraints) and n columns (for the number of variables). Here we let $m_{eq} = 6$, $m_{ineq} = 6$ and $n = 20$. For each constraint (row), random means and variances are drawn from a uniform distribution between 1 and 10. For each constraint, the matrix A is populated by generating random normal variables, where mean and var come from the previously generated means and variances. The first m_eq rows of the matrix A are assigned to the equality constraint matrix A_1, and the remaining rows (i.e., the inequality constraints) are assigned to A_2.

b_1 represents the right-hand side of the equality constraints. The means and variances for b_1 are randomly drawn from uniform distributions with values between -100 and 100 for the means, and between 1 and 100 for the variances. Each b_1 value is generated using a normal distribution. b_2 represents the right-hand side of the inequality constraints. Similar to b_1, the means and variances for b_2 are randomly drawn, and each b_2 value is generated using a normal distribution.

The vector g represents the linear coefficients in the objective function. It is generated randomly from a normal distribution with mean 0 and variance 1. L is an $n \cdot n$ matrix of random values drawn from a normal distribution with mean 1 and standard deviation 2. H is generated using the formula $H = 0.1I + L^T L$, where I is the identity matrix. This ensures that H is **positive semi-definite**, which guarantees the convexity of the objective function.

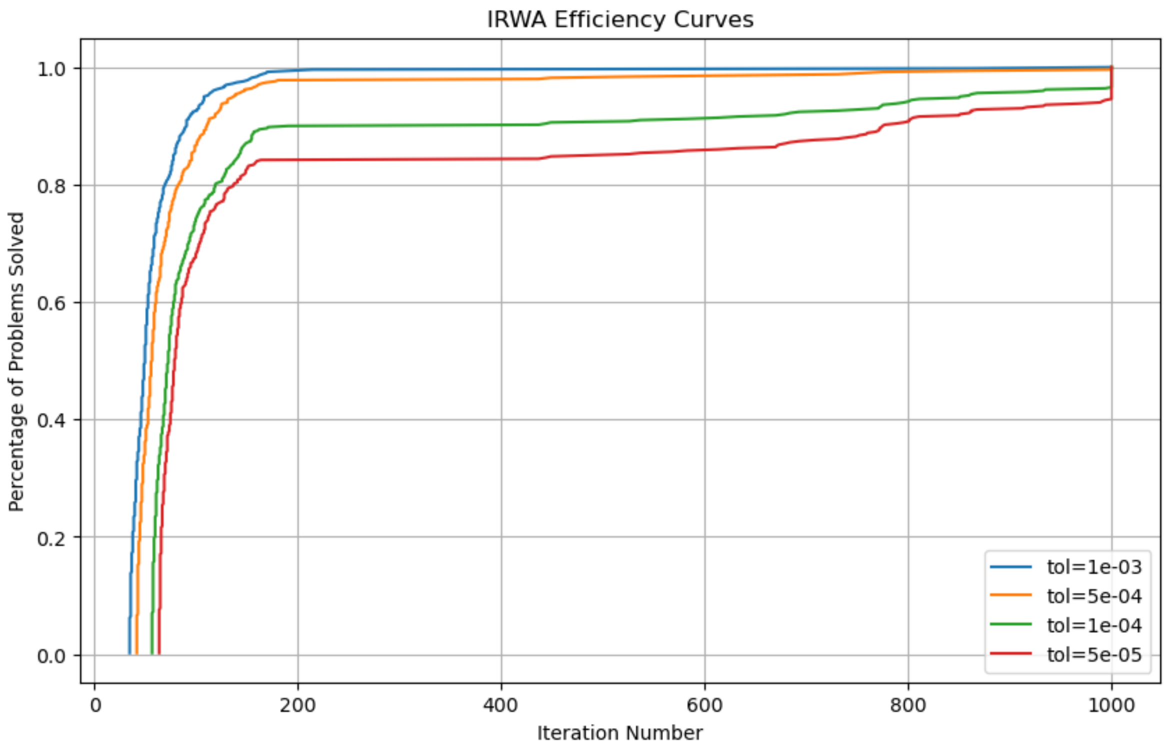

Figure 11: Effect curves for IRWA under different tolerances.

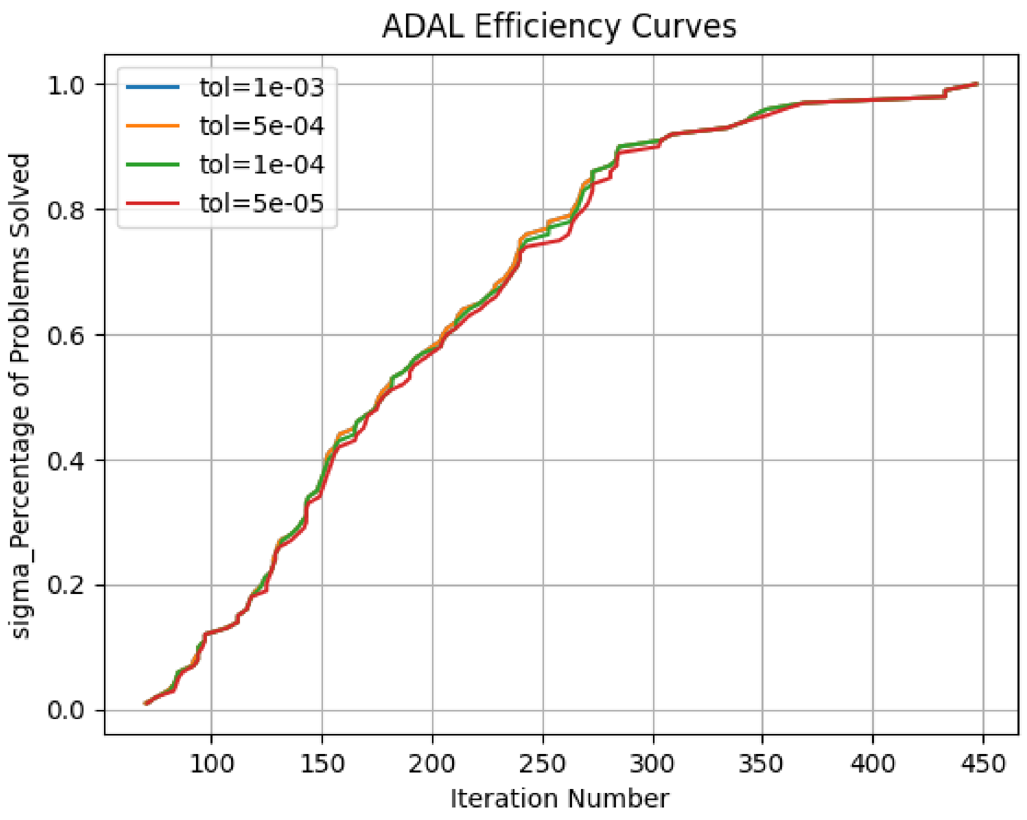

Figure 12: Effect curves for ADAL under different tolerances.

Figure 11 and Figure 12 contains efficiency curves for IRWA and ADAL, which shows the percentage of the problems solved versus the total number of iterations. From the figure, it's easy to learn that when tolerance gets smaller, the efficiency of the algorithm gets lower. IRWA terminated in fewer than 200 iterations with some problems unsolved until large iteration numbers. ADAL required more iterations under the same tolerances.

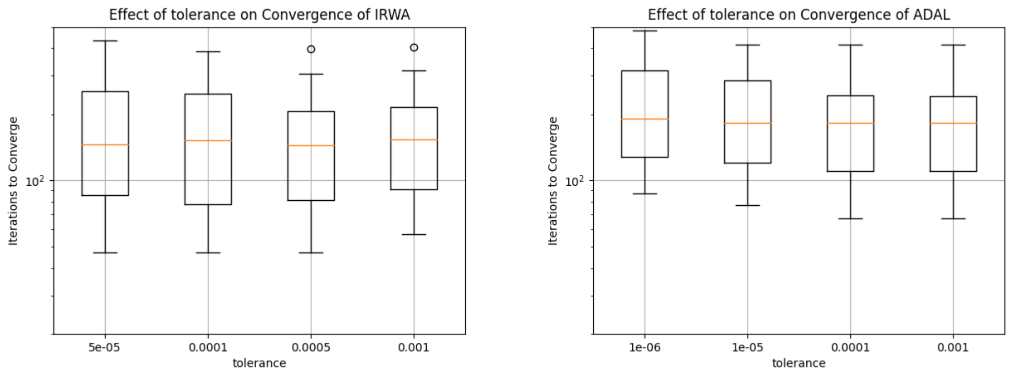

Figure 13: Box plot for IRWA and ADAL under different tolerances.

Figure 13 includes two box plots showing the logarithm of the number of iterations needed by each algorithm for each chosen accuracy level. The results suggest that the methods are largely comparable.

4.4 Analyzing the impact of problem size

In this experiment, we designed 30 test cases, each with progressively larger dimensions. The number of variables, denoted by n, was determined using the formula $n = 200 + 300(j - 1)$, where $j = 1, \ldots, 30$. For each test case, the number of constraints m was set as $m := n/2$. The matrix A was generated using the same procedure as outlined in the preceding experiment.

The vectors b and g were constructed by drawing random samples from a normal distribution. The mean and variance for this distribution were chosen uniformly at random from the ranges $[-200, 200]$ and $[1, 200]$, respectively. For each problem, the matrix H was defined as $H = 40I + LDL^T$, where L was a matrix of size $R^{n\times 8}$ and D was a diagonal matrix. The entries of L were generated using the same approach as for A, while the diagonal elements of D were sampled from an inverse gamma distribution with the probability density function $f(x) = \frac{b^a}{\Gamma(a)} x^{-a-1} e^{-b/x}$. The parameters of the inverse gamma distribution were set to $a = 0.5$ and $b = 1$.

The experimental setup also included the following parameter settings: $\eta := 0.5$, $M := 10^4$, and $\gamma := \frac{1}{6}$. For $j = 1, \ldots, 20$, the initial values ε_{0i} were defined as $\varepsilon_{0i} := 10^2 + 1.3 \ln(j + 10)$, for $i = 1, \ldots, m$, and the parameter μ was specified as $\mu := 500(1 + j)$.

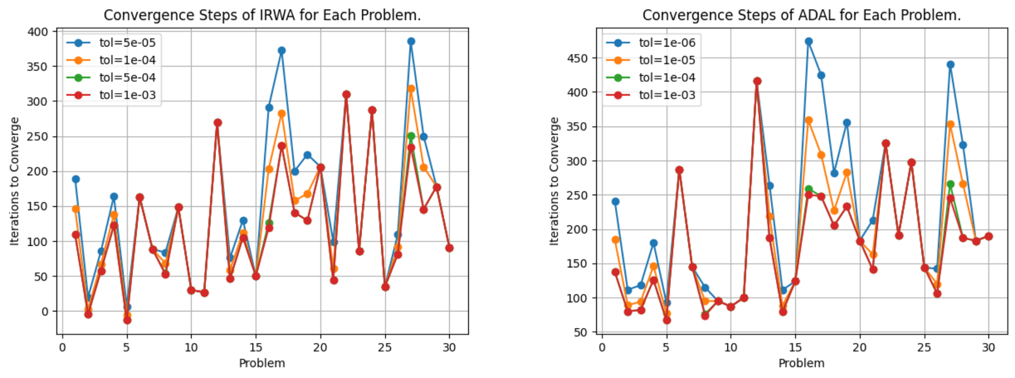

Figure 14: Increase in iterations for different tolerances for IWRA and ADAL as dimension is increased.

In Figure 14, we present two figures showing the number of iterations to convergence versus the dimensions of the variables for the two methods. The figures show that the iterations required for convergence increase as the problem size increases. The two algorithms performed similarly.

5. Conclusion

In this project, we conducted an in-depth exploration of quadratic programming (QP) with a focus on theoretical underpinnings, existing algorithmic approaches, and practical implementations. Through our study of algorithms such as Iterative Reweighted Algorithm (IRWA) and the Alternating Direction Method of Multipliers (ADAL), we analyzed their effectiveness, convergence properties, and performance under various conditions.