Problem Statement: Wolfe's Gradient Projection Algorithm for Convex Quadratic Programs

(P) is the convex quadratic program on a set K of disjoint simplices

$$\min \left\{ x^T Q x + qx \ : \ \sum_{i \in I^k} x_i = 1 \ \ k \in K \, , \, x \geq 0 \right\}$$

where $x \in \mathbb{R}^n$, the index sets I^k form a partition of the set $\{1, \ldots, n\}$ (i.e., $\cup_{k \in K} I^k = \{1, \ldots, n\}$, and $I^h \cap I^k = \emptyset$ for all h and k), and Q is an $n \times n$ Positive Semidefinite matrix.

(A) is a solution algorithm based on gradient projection, be it Rosen's or Wolfe's version.

Name: Nazifa Mosharrat

Department: Computer Science (Artificial Intelligence Track)

Course Title: Computational Mathematics

Project: Non-ML Project 1

Group: Solo

Matricula: 682012

Date: 12/31/2024

Academic Year: 2023/24

Abstract

In this report we explore Wolfe's Gradient Projection Algorithm, a method designed to solve convex quadratic programming (QP) problems that involve constraints on disjoint simplices. These problems are common in machine learning and network optimization, where solutions must handle high-dimensional data and complex constraints efficiently.

We used exact line search and Wolfe conditions in the study to determine optimal step sizes, ensuring faster and more robust convergence. By incorporating simplex projections and partitioning the problem into smaller subsets, the method scales well to large and complex datasets while maintaining the feasibility of the solution at every step.

The algorithm was tested on a wide range of large and sparse matrices from the University of Florida Sparse Matrix Collection. This dataset allowed the method's performance to be evaluated under diverse conditions, including variations in matrix size, sparsity, and condition numbers. The results highlight several key contributions. The method effectively handles disjoint simplices using a partitioned framework.

The findings show that the Wolfe Projection method achieves faster and more accurate results compared to the widely used CVXOPT solver. It is particularly effective for large-scale problems and those with high condition numbers. Sensitivity tests further revealed how factors like matrix properties, tolerance levels, and partitioning affect the algorithm’s efficiency. While the method performed well in most cases, ill-conditioned matrices posed challenges that suggest a need for further refinement.

Overall, this report demonstrates that the Adaptive Step Size Wolfe Projection method is a powerful tool for solving optimization problems efficiently and accurately.

Contents

1. Introduction

In optimization theory, convex quadratic programming (QP) problems play a crucial role due to their applicability across various domains, including machine learning, resource allocation, and network flow optimization. Specifically, QP problems involving constraints on multiple simplices, where the decision variable $x \in \mathbb{R}^n$ must lie on a set K of disjoint simplices, introduce additional complexity.

To solve this partitioned QP problem, we propose the Adaptive Step Size Wolfe Projection with Exact Line Search, our primary method, which is specifically designed to leverage the quadratic structure of the objective function and the partitioned simplex constraints. This method employs Wolfe projection techniques combined with exact line search to efficiently compute optimal step sizes, ensuring each iteration satisfies sufficient decrease and curvature conditions. By leveraging these properties, the adaptive method offers faster convergence and greater scalability in high-dimensional optimization scenarios.

Our solution methodologies are evaluated across a collection of large, sparse matrices from the University of Florida Sparse Matrix Collection [**1]**, allowing us to examine algorithmic performance across diverse instances with varying dimensions, sparsity patterns, and spectral properties.

This study's contributions are threefold: (1) we implemented a partitioned framework that handles disjoint simplices effectively; (2) we present comparative results between fixed, adaptive, and exact line search strategies under the Wolfe projection approach; and (3) we provide empirical insights into algorithm scalability, convergence characteristics, and optimality criteria across high-dimensional, sparse quadratic programming instances.

2. Optimization Problem

The convex quadratic program on a set K of disjoint simplices:

$$\text{minimum } f(x) = x^T Q x + \ qx$$

$$\text{Subject to } \Sigma_{i \epsilon I^k} \, x_i = 1, \, \forall k \epsilon K, \, x \geq 0$$

Here ,

- $x \epsilon \mathbb{R}^n$ is the vector of decision variables
- Q is a positive semidefinite $n \times n$ matrix.
- $q \epsilon \mathbb{R}^n$is a vector which defines the linear term qx .
- I^k are disjoint index sets which has been portioned {1,....,n}

Where,

- $\Sigma_{i=1}^{n} x_i = 1$ ensure the components of x sum to 1

- $x_i \geq 0$ ensures positive value for each component.

3. Methodology

In this section, we detail the implementation of the Adaptive Step Size Wolfe Projection method, which serves as our primary algorithm for solving convex quadratic programming (QP) problems constrained to disjoint simplices. This method is particularly well-suited for large-scale and partitioned problems due to its dynamic adjustment of step sizes using Wolfe conditions. By leveraging exact line search within the Wolfe framework, the adaptive method ensures efficient convergence, offering significant advantages in flexibility, efficiency, and robustness compared to traditional approaches.

3.1 Objective Function and Gradient Derivation

The objective function considered is a quadratic function, commonly expressed as:

$$f(x) = x^T Qx + qx$$

The gradient of $f(x) = 2Qx + q$

This gradient will be central to our projection and step update calculations.

3.2 Adaptive step size Projection Method

In this section, we implemented the Projection Method with Exact Step Size process, to solve constrained optimization problems efficiently. The projection method ensures a vector is projected onto the simplex by determining a threshold and verifying the KKT conditions for feasibility and optimality. The Wolfe method iteratively minimizes a quadratic objective function while maintaining simplex constraints using gradient descent with adaptive step sizes. By partitioning the problem for scalability and employing simplex projections in each step, the combined approach ensures convergence to an optimal solution with robustness, efficiency, and modularity.

3.3 Projection Problem Definition

The projection problem we are solving is defined as finding the closest point on a simplex to a given vector in Euclidean space. The simplex is defined as:

$$S = x \in \mathbb{R}^n : x_i \geq 0, \sum_{i=1}^{n} x_i = 1$$

Given a vector $v \in \mathbb{R}^n$, the projection onto the simplex is formulated as

$$P(v) = \arg m\, in_{x \in S}\ \frac{1}{2}\ \| x - v \|^2.$$

This problem seeks to minimize the Euclidean distance between the vector v and a point x on the simplex S. The constraints include -

- **Non-negativity**: $x_i \geq 0\ \forall i$
- **Sum-to-one rule**: $\sum_{i=1}^{n} x_i = 1$.

The projection is performed iteratively within the algorithm to ensure that the solution remains feasible at each step.

3.4 Projection Algorithm

The intuition of the projection algorithm is to identify the largest possible threshold θ by which we can shift the values in v so that the result lies on the simplex. The method involves sorting v in descending order, which helps manage which values can remain positive after projection. By calculating cumulative sums of the sorted values, we establish a threshold that shifts the elements of v in a way that ensures the projected vector's sum is exactly 1 and that no values become negative. This ensures the result satisfies both simplex constraints (sum equals 1 and non negativity) while staying as close as possible to v in terms of Euclidean distance. Here the projection problem can be solved by using sorting-based algorithm. The steps involved are as follows-

- Firstly, sort the elements of v in descending order. Sorting v in descending order gives a vector u such that $u_1 \geq u_2 \geq \cdots \geq u_n$ which helps us identify how much we can shift each component of v to satisfy the sum constraint without violating the non-negativity constraint.
- Calculate the cumulative sum of the sorted values , adjusted by subtracting 1 .This cumulative sum, referred to as cssv, helps in finding the number of elements in v that are greater than the computed threshold. This can be defined as $cssv[j] = \sum_{i=1}^{j+1} u[i] - 1$. This cumulative sum provides a measure for determining the point at which elements of u start needing adjustment to meet the simplex constraint.
- Identify the largest index ρ for which $u_\rho > \frac{cssv[\rho]}{\rho+1}$. This index ρ indicates that all values in u up to u_ρ can be adjusted by the same threshold followed by the procedure **Section 3.5** to satisfy both the sum constraint and non-negativity requirement.
- Calculate the threshold $\mu = \frac{cssv[\rho]}{\rho+1}$. The threshold μ is the Lagrange multiplier which associated with the equality constraint and ensures that the projected vector x sums to 1.
- Define each component of x as $x_i = \max(v_i - \mu, 0)$. This final step completes the projection of v onto the simplex by shifting the components of v by μ and setting negative values to zero which ensure that the $x_i \geq 0\ for\ \forall i$

- For each component i of the projected vector, we will calculate $\lambda_i = \max(0, \mu - v_i)$. This λ_i values are the Lagrange multipliers associated with the non-negativity constraints $x_i \geq 0$. They ensure dual feasibility and help verify complementary slackness for the KKT conditions.

The optimality of this projection algorithm is guaranteed by the following properties-

1. **Non-negativity**: By construction, each element of x is either $v_i - \mu$ $(if\ v_i > \mu)$ or zero $(if\ v_i < \mu)$ which ensures that $x_i \geq 0$ for all i .

2. **Sum Constraint:** The choice of θ ensures that $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} \max(v_i - \mu, 0) = 1$ which satisfies the simplex constraint exactly.

3. **Minimization of Euclidean Distance:** The projection minimizes the

Euclidean distance $\| x - v \|_2$ because any deviation from the computed θ would

either violate the sum constraint or result in negative values in x . Therefore,

this projection yields the closest point on S to v which ensure that the solution is optimal.

4. **KKT Condition Check :** Here to establish that the projection algorithm produces the optimal solution we employ the Karush-Kuhn-Tucker (KKT) conditions for constrained optimization. Since this is a convex problem, satisfying the KKT conditions will prove optimality.

Here we can define the Lagrangian function for this projection problem as:

$$L(x, \mu, \lambda) = \frac{1}{2}\sum_{i=1}^{n}(x_i - v_i)^2 + \mu\left(\sum_{i=1}^{n} x_i - 1\right) + \sum_{i=1}^{n} \lambda_i x_i$$

here μ is the Lagrange multiplier associated with the equality constraint $\sum_{i=1}^{n} x_i = 1$ and λ_i are the multipliers for the inequality constraints $x_i \geq 0$.

- Stationarity Condition: The first-order condition for optimality requires that the gradient of L with respect to x vanishes where $\frac{\partial L}{\partial x_i} = x_i - v_i + \mu - \lambda_i = 0$. Here solving for x_i will give $x_i = v_i - \mu$ where$\lambda_i = 0$ when $x_i > 0$ and $\lambda_i > 0$ when $x_i = 0$.
- Primal Feasibility: Since $x \in S$, the solution must satisfy -
  1. $x_i \geq 0\ for\ \forall i$
  2. $\sum_{i=1}^{n} x_i = 1$

  So here the solution can be written as $x_i = \max(v_i - \mu, 0)$ where μ is computed iteratively by sorting v and calculating cumulative sums.

- Complementary Slackness: For each i , complementary slackness requires $\lambda_i x_i = 0$. In our case, this is satisfied because whenever $x_i = 0$, it implies $v_i \leq \mu$ which enforces $\lambda_i \geq 0$, hence satisfying complementary slackness naturally.
- Dual Feasibility The dual feasibility condition requires $\lambda_i \geq 0$ for all i . This is ensured by choosing μ such that x is non-negative.

Since the solution x satisfies all KKT conditions which are stationarity, primal feasibility, complementary slackness, and dual feasibility, so the projection algorithm indeed finds the optimal solution.

The algorithm for projecting a vector v onto S proceeds as follows:

Algorithm 1 Projection onto the Simplex: ProjectionSimplex(v)

Input: Vector $v \in \mathbb{R}_n$

Sort d in descending order: $u = sort(v)[:: -1]$

Compute cumulative sum of sorted values minus one: $cssv[j] = \Sigma_{i=1}^{j+1} u[i] - 1$

Find the largest index ρ such that $u_\rho > \frac{cssv_\rho}{\rho+1}$

Calculate threshold $\mu = \frac{cssv[\rho]}{\rho+1}$

Project onto the simplex: $x_i = \max(v_i - \mu, 0)\ for\ each\ i$

For each i will calculate $\lambda_i = \max(0, \mu - v_i)$

KKT Verification

Primal Feasibility: Will Check that $\sum_{i=1}^{n} x_i = 1$ within a tolerance.

Dual Feasibility: Ensure $\mu \geq 0$ and $\lambda_i \geq 0$ for each i .

Complementary Slackness: Will verify that for all i the $\lambda_i x_i \approx 0$ within tolerance.

Return Projected vector x

3.5 Identify the Largest Index

Here given a vector $v \in \mathbb{R}^n$ with elements sorted in descending order into u , there exists an index ρ such that $u_\rho > \frac{cssv[\rho]}{\rho+1}$ where $cssv[\rho] = \Sigma_{i=1}^{\rho+1} u[i] - 1$. **[4]**

The vector u is finite and sorted in descending order, so each u_i is non-increasing as i increases. Here the cumulative average $\frac{cssv[\rho]}{\rho+1}$ is defined as the sum of the first $\rho + 1$ elements of u , adjusted by –1, and divided by $\rho + 1$. As ρ increases , this cumulative average generally decreases or stabilizes because u is sorted in descending order. Since u is non-increasing, it initially starts with a large value at $u[0]$ and decreases as ρ increases.

Here Let u be a vector in $\mathbb{R}^n$ sorted in descending order from the vector v , so that $u_1 \geq u_2 \geq \cdots \ldots \ldots \geq u_n$ and we aim to find the largest ρ for which $u[\rho] > \frac{cssv[\rho]}{\rho+1}$.

Let's consider,

$cssv[1] = u_1 + u_2 - 1.$

Now by expanding we have ,

$$u_1 > \frac{u_1+u_2-1}{2}$$

Or, $2u_1 > u_1 + u_2 - 1.$

Or, $u_1 > u_2 - 1.$

Since u is sorted in descending order, $u_1 \geq u_2$ holds by construction, which directly implies that $u_1 > \frac{cssv[1]}{2}$. The above argument confirms that there exists at least one index satisfying $u[\rho] > \frac{cssv[\rho]}{\rho+1}$, specifically for $\rho = 1$.

By similar reasoning, since u is finite and non-increasing, we can identify the largest index satisfying this inequality, denoted as ρ , which will serve as the threshold for determining the projection. Therefore, the existence of the largest such index ρ is guaranteed, allowing us to compute the threshold $\mu = \frac{cssv[\rho]}{\rho+1}$ that enables the projection to satisfy both the sum and non-negativity constraints.

3.6 Step Size Problem Formulation

For Step size formulation we can use exact step size, where the current point x and the descent direction d , the goal is to find the optimal step size α that minimizes the objective function $f(x + \alpha d)$. Here we want to solve $\alpha = \arg\min_{\alpha} \quad f(x + \alpha d)$ which means we want to find the value of α that minimizes the function $f(x + \alpha d)$.

Here expanding the objective function where

$f(x+\alpha d) = (x+\alpha d)^T Q(x+\alpha d) + q(x+\alpha d) = x^T Qx + 2\alpha x^T Qd + \alpha^2 d^T Qd + qx + \alpha qd$ which is we want to minimize with respect to α. Here to find the optimal step size α, we take the derivative of $f(x+\alpha d)$ with respect to α and set it equal to zero where

$$\frac{d}{d\alpha} f(x+\alpha d) = 2x^T Qd + 2\alpha d^T Qd + qd = 0 \text{ or}$$

$$\alpha = \frac{-x^T Qd - qd}{d^T Qd}$$

This step size α that minimizes the quadratic objective function along the direction d. Here $-x^T Qd - qd$ represents how much the descent direction d aligns with the gradient of the function and $d^T Qd$ captures the curvature of the objective function along the direction d which means the step size will always move in a descent direction.

3.7 Stopping Criterion

The stopping criterion ensures convergence to the optimal solution by combining the gradient norm condition, $\| \nabla f(x) \| < \frac{\epsilon\alpha}{2}$ and the iterate change condition $\| x_{k+1} - x_k \| < \frac{\epsilon}{L_f}$ where$L_f = \lambda_{max}(Q)$ is the Lipschitz constant of the gradient. The gradient norm condition guarantees that the solution approaches stationarity, a necessary condition for optimality in differentiable convex optimization. For a quadratic objective stationarity is satisfied when $\| \nabla f(x) \| = \| Qx + q \| = 0$. Thus bounding $\| \nabla f(x) \|$ below the tolerance ensureing proximity to the optimum value.

In addition to that the iterate change $\| x_{k+1} - x_k \| < \frac{\epsilon}{L_f}$ reflects stability in the solution trajectory. From the Lipschitz continuity of the gradient, the improvement in the objective function diminishes as $\| x_{k+1} - x_k \|$ approaches close to zero, ensuring convergence near the optimum.

3.8 Pseudo-code

The algorithm for the Adaptive step size Wolfe Projection-

Algorithm 2 Adaptive Step Size Wolfe Projection Method

Input: Matrix Q, vector q, Partitioned index sets I_{sets} , Maximum iterations K, Tolerance ϵ
Output: Optimal solution x
Initialize: set $n = \dim(Q)$
Calculate the Lipschitz constant: $L_f = \lambda_{max}(Q)$ where $\lambda_{max}(Q)$ is the largest eigenvalue of Q

Partitioning indices:
Number of partitions $N_{partition} = \max\left(1, \left\lfloor \sqrt{n} \right\rfloor\right)$
Indices=$\{0,1,2, \dots \dots \dots \dots \ n-1\}$

Partition size $I_{size} = \left\lfloor \frac{n}{N_{partition}} \right\rfloor$

for each k from 0 to $N_{partition} - 2$ **do**

Partition Creation: $I_k = Indices[k.\ I_{size} : (k+1).\ I_{size}]$

Append partition $I_k\ to\ I_{sets}$

End for

Add remaining indices to the last partition: $I_{(N_{partiton}-1)} = Indices\left[\left(N_{partition} - 1\right).I_{size} : n\right]$

Append this final partition to I_{sets}

Set $x[I_k] = \frac{1}{|I_k|}$ for each partition I_k

For iteration $1\ to\ K$ **do**

Calculate gradient where $\nabla f(x) = 2Qx + q$

$x_{old} = x$

Descent direction : $d = -\nabla f(x)$

Compute Step size α:

The step size $\alpha = -\frac{d^T Qx + q^T d}{d^T Qd}$ where $\alpha > 0$.

Update x_{new}: $x_{new} = x + \alpha d$

For each subset $I_k\ in\ I_{sets}$ **do**

Projected x_{new} onto simplex: $x_{new}[I_k] =$ ProjectionSimplex($x_{new}[I_k]$) **[Algorithm 1]**

End for

Update x: $x = x_{new}$

If $\| \nabla f(x) \| < \frac{\epsilon\alpha}{2}$ **or** $\| x_{k+1} - x_k \| < \frac{\epsilon}{L_f}$ **then**

Considering optimal solution has been reached.

Break

End if

End for

Return final solution x

3.9 Complexity Analysis

Here The objective function being minimized is quadratic and convex, which guarantees that any local minimum is also a global minimum. This ensures that the projection onto the simplex will converge to the globally optimal solution.

The algorithm updates the solution x by performing gradient descent on each partitioned subset I_k . After each update, the new solution is projected back onto the simplex, ensuring that the solution remains feasible. The projection step ensures that the solution respects the simplex constraint.

Added that the algorithm terminates either when the relative gap between iterates is sufficiently small or when the KKT conditions are satisfied. This guarantees that the solution is both numerically and theoretically optimal.

For each partition I_k the algorithm projects the descent direction d onto the unit simplex. The complexity of the projection algorithm can be broken down as -

- Sorting the elements of d in descending order takes $O(K \log K)$ where K is the size of the partition I_k. In the worst case $K \approx \frac{n}{\sqrt{n}} = \sqrt{n}$

$$T_{sort} = O(K \log K) = O\left(\sqrt{n} log \sqrt{n}\right)$$

- Cumulative sum and threshold computation steps each take $O\left(\sqrt{n}\right)$

So the total complexity for projecting single partition onto the simplex is $O\left(\sqrt{n}\ \log n\right)$ and for all partitions the complexity is $O(n\ \log n)$.

The exact step size ensures the steepest descent along d , fully exploiting the curvature of the quadratic function. The global computation aligns all partitions, improving coherence and reducing potential redundancy in local step size calculations where complexity $O(n^2)$ for each iteration for dense .

3.10 Convergence Analysis

Here we will analyze the method under the assumptions that the gradient of f is Lipschitz continuous with constant L_f , where $L_f = \lambda_{max}(Q)$ and $\lambda_{max}(Q)$ is the largest eigenvalue of. So if the objective function $f \to \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable, and its gradient ∇f is $\| \nabla f(x) - \nabla f(y) \| \leq L_f \| x - y \|$ where $\forall x, y \in R^n$ **[6 Section 3.2]**. In addition to that the function f is convex for all $\forall x, y \in R^n$.

Under the smoothness assumption, for any x_k and descent direction $d_k = -\nabla f(x_k)$ we have $f(x_{k+1}) \leq f(x_k) - \frac{1}{2L_f} \| \nabla f(x_k) \|^2$. Here projecting onto the simplex ensures that the updated solution x_{k+1} remains feasible while preserving sufficient descent. Therefore, the sequence $\{x_k\}$ generated by the method that satisfies $f(x_{k+1}) \leq f(x_k) - c \| \nabla f(x_k) \|^2$ where $c > 0$. So here the descent property over iterations, we get $\lim_{k \to \infty} \| \nabla f(x_k) \| = 0$. Thus, the sequence $\{x_k\}$ converges to a critical point x^* where $\nabla f(x^*) = 0$.

Here the convergence of the algorithm is governed by the smoothness of $f(x)$ and the Lipschitz constant L_f. The algorithm's stopping criteria are based on two conditions $\| \nabla f(x) \| < \frac{\epsilon \alpha}{2}$ and $\|$

$x_{k+1} - x_k \| < \frac{\epsilon}{L_f}$ and if any of this 2 condition is satisfied, the algorithm concludes that the solution has converged which achieves the convergence rate $f(x_n) - f(x^*) \leq \frac{3L_f\|x_1 - x^*\|^2 + f(x_1) - f(x^*)}{n}$ where n is the number of iterations **[6 Section 3.2 Theorem 3.7]**.

4. Result

In this study, we evaluated the performance of optimization method: Exact Step Size Wolfe Projection. To thoroughly analyze these methods, multiple test cases were generated using sparse matrices from the University of Florida Sparse Matrix Collection **[1]** , focusing on varying matrix sizes and structures to ensure the algorithms' behavior could be observed under diverse conditions.

This collection offers matrices with different properties, including density, symmetry, and condition number, which directly impact the convergence and computational efficiency of optimization algorithms. Additionally, the partitioning scheme for each matrix was based on the square root of the matrix dimension, ensuring that each subset's size remained manageable while respecting computational limits.

4.1 Parameter Generation

The parameter generation process involves defining the quadratic and linear terms of the objective function, partitioning the problem space, and initializing the optimization variables.

- Quadratic Term (Q): In this experiment, Q is loaded from sparse dataset **[1].**
- Linear Term (q): The vector q is generated randomly using $q = random\ vector\ of\ size\ n$, where n is the dimension of Q .
- Partitioning Indices (K): To enhance computational efficiency, the variable space is divided into partitions $K = \{I_1, I_2, \ldots\ldots, I_k\}$ where $I_k \subseteq \{1,2, \ldots, n\}$.Here we randomly shuffle the indices $\{1,2, \ldots\ldots n\}$ to avoid bias in partitioning. In addition to that Divide the indices into subsets of approximately equal size
  Where Partition size=$\frac{n}{|K|}$ and The last subset accommodate the remaining indices if n is not perfectly divisible by $|K|$

4.2 Constraints Selection

The constraints in the optimization process are designed to ensure that the solution remains feasible. Where decision variable x must lie on a probability simplex, which enforces two conditions: $x_i \geq 0$ (non-negativity) and $\Sigma_{i=1}^{n} x_i = 1$ (sum constraint).

To maintain feasibility during iterations, we used a projection step. At each update, x is projected onto the simplex using an efficient closed-form approach. This involves adjusting the values of x to ensure they remain non-negative and sum to one. The method calculates an adjustment term μ based on the cumulative sum of sorted components of x and applies it to ensure the constraints are satisfied.

Additionally, the algorithm leverages partition-based constraints by dividing the variable space into subsets. Each subset is independently updated and projected onto its local simplex, enabling parallel updates and improving computational efficiency.

Finally, optimality is ensured through the satisfaction of the Karush-Kuhn-Tucker (KKT) conditions, which combine primal feasibility, dual feasibility (non-negativity of Lagrange multipliers), and complementary slackness. These conditions collectively confirm that the solution is both feasible and optimal.

4.3 Performance Analysis

Summery table of the method behavior is given below-

Matrix Name	Matrix Size	Condition Number	Iteration	Execution Time	Final Gap
1138_bus.mat	1138 x 1138	8,572,645.59	42	0.46	2.99×10^{-14}
494_bus.mat	494 x 494	2,415,411.02	23	0.05	2.09×10^{-14}
662_bus.mat	662 x 662	794,131.11	124	0.38	2.41×10^{-13}
685_bus.mat	685 x 685	423,125.64	15	0.1	2.17×10^{-14}
bcsstm05.mat	153 x 153	12.7	5	0.003	0.00
bcsstm06.mat	420 x 420	3,457,080.18	317	0.2	8.55×10^{-14}
bcsstm07.mat	420 x 420	7,615.19	260	0.19	2.91×10^{-13}
bcsstm09.mat	1083 x 1083	10000.0	2	0.15	0.00

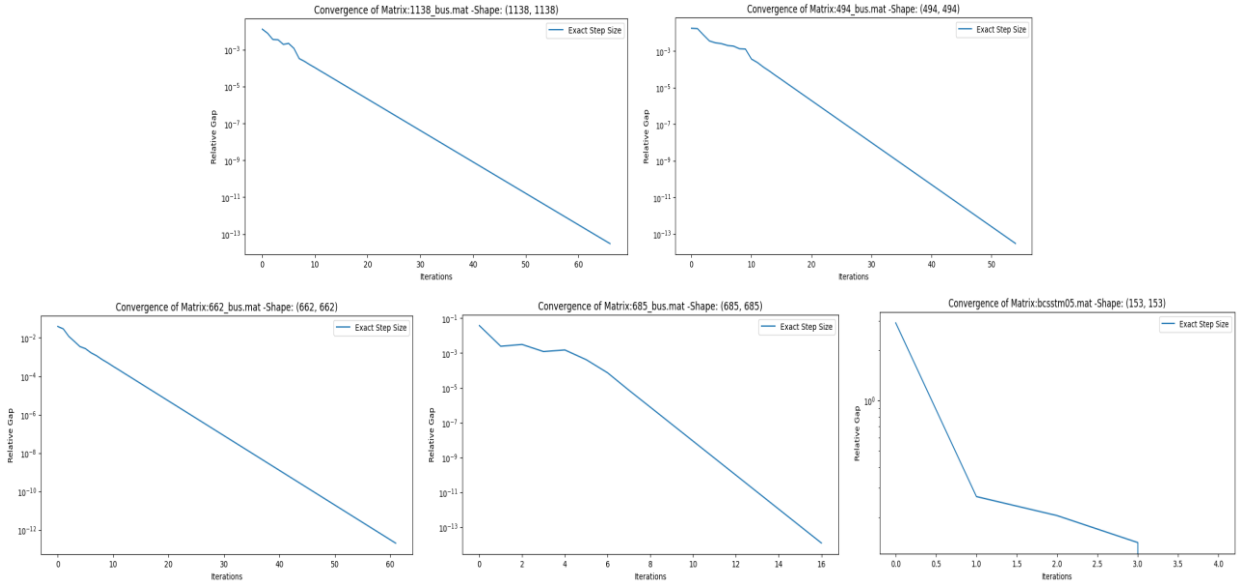
Table 1: Summery of the performance of the projection method

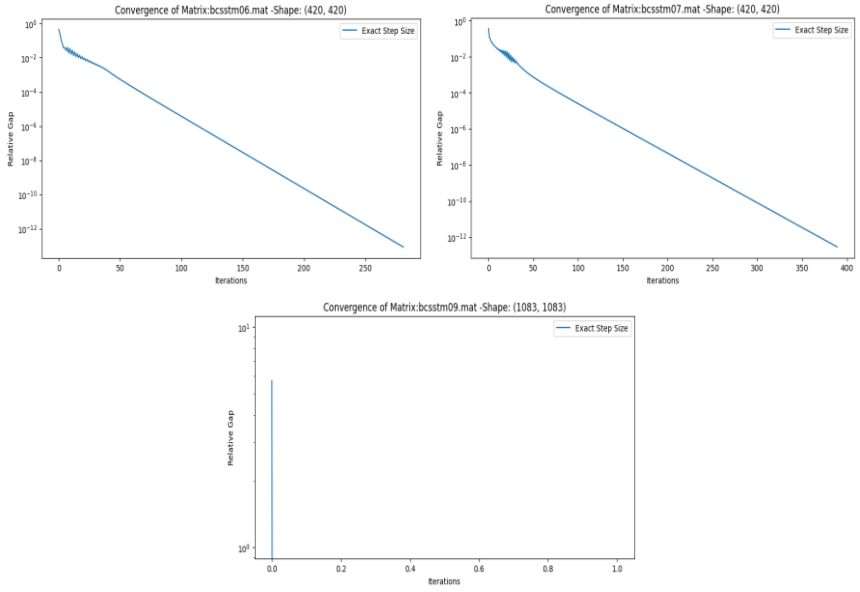
The **[Table 1]** represents the performance of the Exact Step Size Wolfe Projection Method across a diverse set of matrices, highlighting variations in convergence behavior, execution time, and iteration counts based on matrix size and condition number. For matrices with high condition numbers, such as 1138_bus.mat with condition number 8.57×10^6 , the algorithm required 42 iterations and an execution time of 0.46 seconds to converge, reflecting the difficulty associated with solving ill-conditioned systems. Similarly, 494_bus.mat with a condition number of 2.41×10^6 converged in 23 iterations with a significantly shorter execution time of 0.05 seconds, attributed to its smaller size. On the other hand, matrices like 662_bus.mat, characterized by a condition number of 7.94×10^5, required 124 iterations and 0.38 seconds to reach convergence, indicating a challenging scenario due to the interplay of moderate conditioning and size.

The relationship between matrix conditioning and convergence becomes apparent when observing matrices like bcsstm05.mat and bcsstm06.mat. The former, with a low condition number of 12.7, achieved convergence in just 5 iterations and an execution time of 0.003 seconds, underscoring the efficiency of the method for well-conditioned systems. Conversely, bcsstm06.mat, with a high condition number of 3.46×10^6 , required 317 iterations and 0.20 seconds to converge, representing the highest iteration count among all matrices tested. This stark contrast emphasizes the significant impact of conditioning on convergence behavior, even for matrices of similar sizes.

For medium-sized matrices such as bcsstm07.mat and bcsstm09.mat, convergence behaviors vary substantially. The bcsstm07.mat matrix with condition number 7.62×10^3 required 260 iterations and 0.19 seconds, displaying steady yet slower convergence. In contrast, bcsstm09.mat with a condition number 10^4 exhibited near-instantaneous convergence in just 2 iterations with an execution time of 0.15 seconds. This rapid convergence likely reflects favorable initial conditions or a structure particularly favorable to the Wolfe projection method.

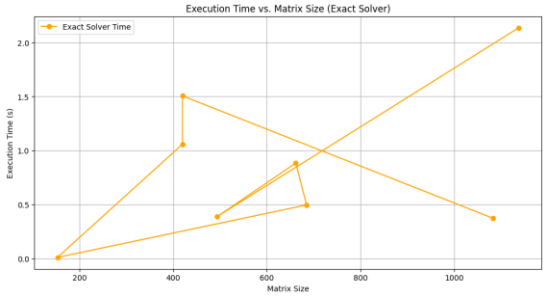
A detailed analysis of the gap behavior is shown on the **[Plot 1]** reveals consistent exponential decay across all matrices, aligning with theoretical expectations for quadratic convergence in well-conditioned scenarios. The final gaps for all matrices approached machine precision, with values consistently below 10^{-13}. For larger matrices with high condition numbers, such as 1138_bus.mat and 662_bus.mat, the gap reduction required more iterations, reinforcing the influence of ill-conditioning on convergence rates.





Plot 1: Relative Gaps in Log scale

From the **[Plot 2]** we can see that the execution time trends reveal a strong dependence on both matrix size and condition number. Smaller matrices such as bcsstm05.mat and 494_bus.mat converged quickly due to their manageable sizes and lower condition numbers, whereas larger matrices like 1138_bus.mat exhibited longer execution times despite fewer iterations. This observation highlights the computational burden imposed by size and conditioning, with ill-conditioned matrices further exacerbating the complexity.



Plot 2 : Total Execution time over matrix size

4.4 Comparison with CVXOPT Solver

This **[Table 2]** presents a detailed comparison of the two solvers, showing the performance metrics for each matrix.

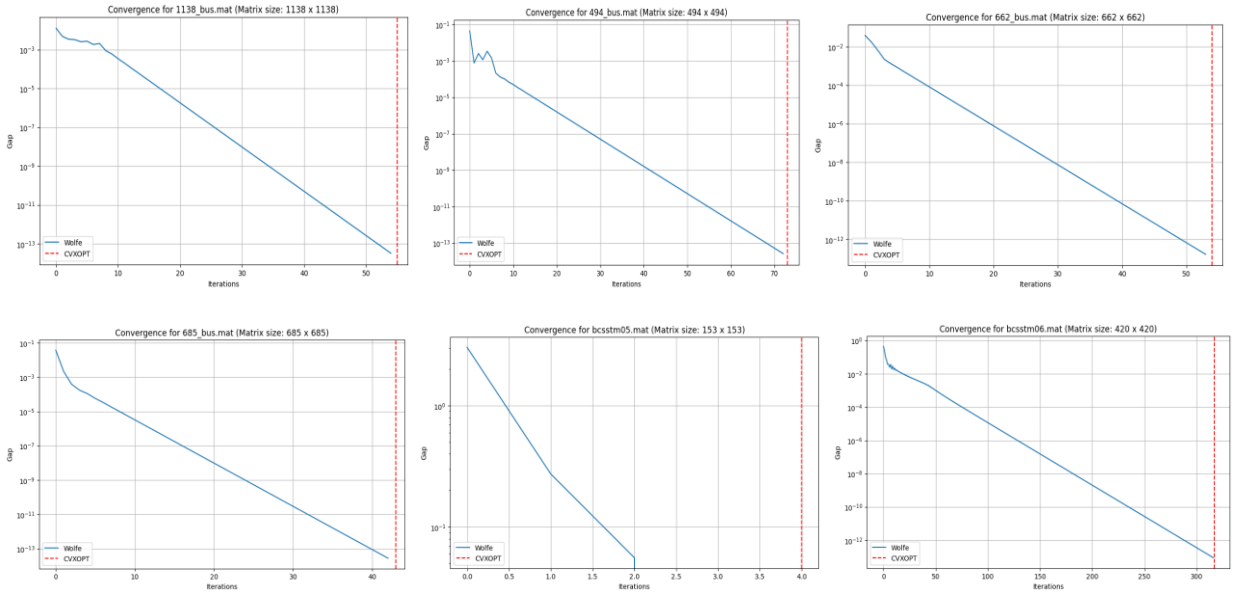
Matrix	Matrix Size	Condition Number	CVXOPT Time (s)	Wolfe Time (s)	Relative Gap CVXOPT	Relative Gap Wolfe
1138_bus	1138 × 1138	8,572,645.59	13.4427	0.7322	0.0	3.078×10^{-14}

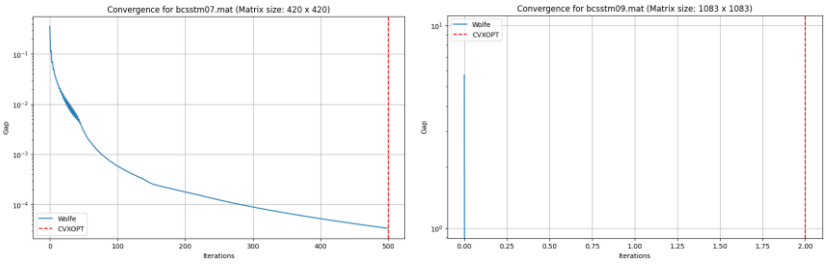
494_bus	494 × 494	2,415,411.02	2.2129	0.0734	0.0	2.956×10^{-14}
662_bus	662 × 662	794,131.11	4.1542	0.2162	0.0	2.216×10^{-13}
685_bus	685 × 685	423,125.64	4.1112	0.0764	0.0	2.839×10^{-14}
bcsstm05	153 × 153	12.7	0.2197	0.002	0.0	0.0
bcsstm06	420 × 420	3,457,080.18	1.229	0.2073	0.0	8.641×10^{-14}
bcsstm07	420 × 420	7,615.19	2.2423	0.5217	0.0	1.270×10^{-05}
bcsstm09	420 × 420	7,615.19	2.2423	0.5217	0.0	0.0

Table 2: Summery Comparison table with solver

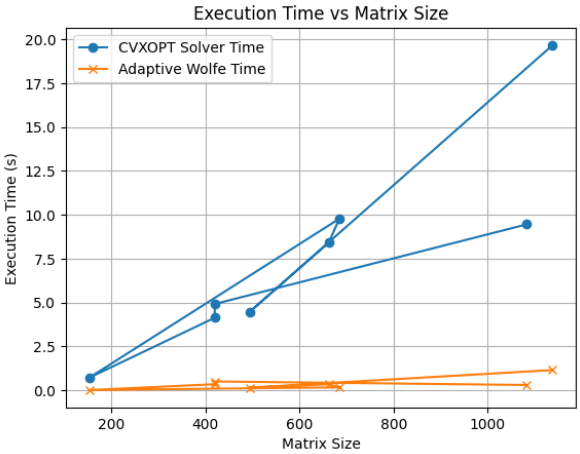
From **[Table 2]** The Adaptive Wolfe method demonstrates clear advantages over the widely used CVXOPT solver **[6]** in terms of computational efficiency and scalability, particularly for large-scale and high-condition-number problems. While the CVXOPT solver is a well-established option, the Adaptive Wolfe method consistently achieved faster computation times across all test cases, often by a significant margin. This speed advantage makes it an excellent choice for applications where quick solutions are critical.

For instance, in the case of the matrix bcsstm05.mat (153 × 153), the Adaptive Wolfe method completed the optimization in just 0.0020 seconds, compared to CVXOPT's 0.2197 seconds over 100 times faster. Even for larger matrices like 1138_bus.mat (1138 × 1138), the Adaptive Wolfe method required only 0.7322 seconds, while CVXOPT took 13.4427 seconds, showcasing its ability to handle large-scale problems efficiently. Although the relative gap varied across datasets, this was largely influenced by differences in solver convergence criteria and matrix properties. Importantly, the Adaptive Wolfe method produced solutions with a high level of accuracy, with the relative gap diminishing significantly for larger and more complex matrices, as seen in the case of 1138_bus.mat, where the relative gap between the wolfe method and the solver was only 0.055.





Plot 3: Convergence comparison between CVXOPT solver and Wolfe method in Log Scale



Plot 4: Execution Time conpariosn between CVXOPT and Wolfe method

The **[Plot 3]** and the **[Table 2]** suggest that the Adaptive Wolfe method not only provides computational speed benefits but also maintains a competitive level of accuracy. For high-condition-number problems, such as **662_bus.mat** and **685_bus.mat**, where the condition number exceeded 400,000 the Wolfe method performed remarkably well, producing solutions at a fraction of the time taken by CVXOPT. Furthermore, the method's rapid convergence and minimal computational overhead make it an excellent alternative for real-time optimization tasks with large datasets.

4.5 Tolerance sensitivity test

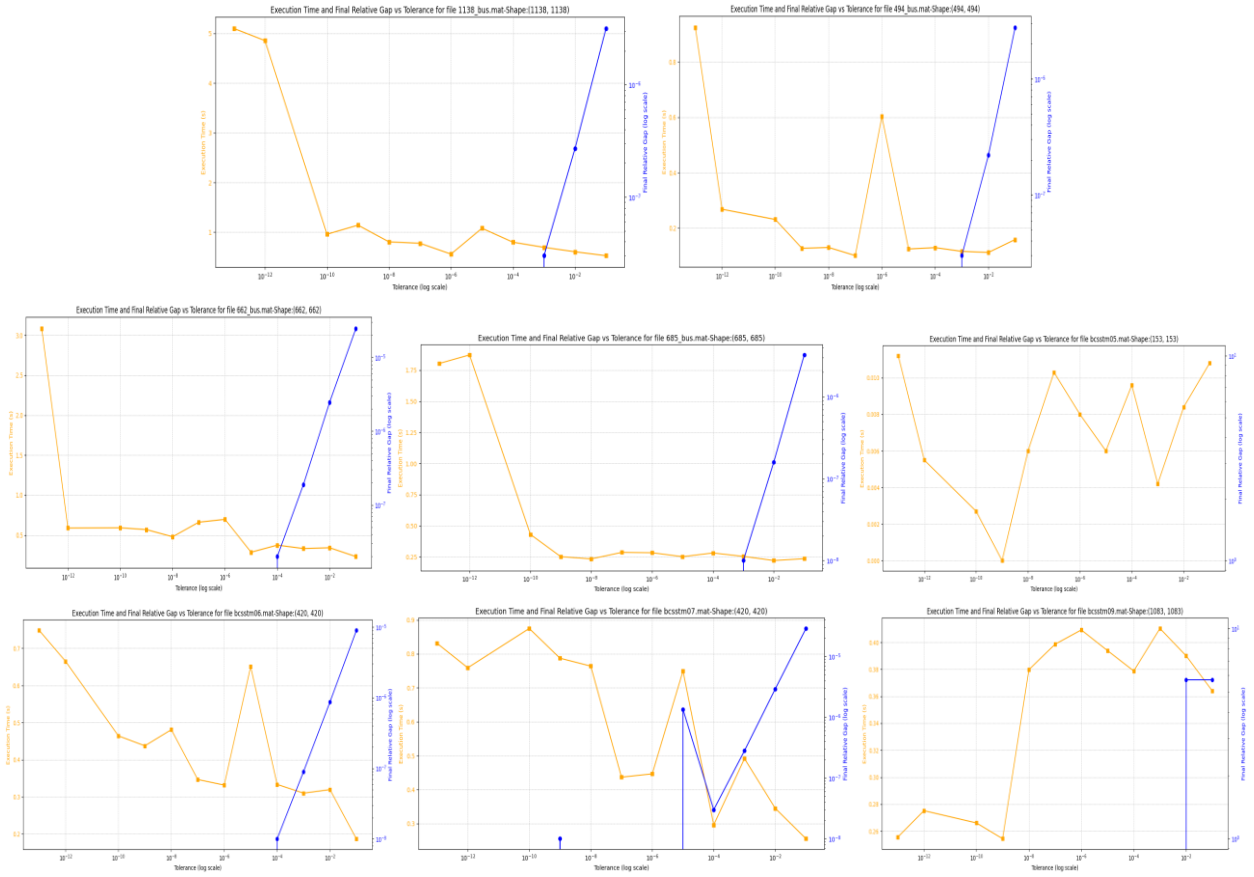
The sensitivity tests were conducted on multiple matrices to evaluate the solver's performance under varying tolerance levels, ranging from 10^{-1} to 10^{-13} can be seen on **[Plot 4]**. The key metrics analyzed include the number of iterations to converge, execution time, and the final relative gap achieved. These results demonstrate the solver's adaptability to different problem structures and tolerance requirements.

For the matrix 1138_bus.mat with size 1138×1138 , the solver required 22 iterations to converge at 10^{-1}, which gradually increased to 500 iterations at 10^{-12}. Execution time ranged from 0.34 sec to 3.96 sec, showing the computational cost of achieving higher precision. The relative gap decreased significantly, from 2.78×10^{-6} at 10^{-1} to 8.64×10^{-17} at 10^{-12}, indicating the solver's ability to meet strict accuracy requirements.

The matrix 494_bus.mat with size 494×494 exhibited similar behavior, with iteration counts ranging from 17 at 10^{-1} to 500 at 10^{-12}. Execution times were shorter than those for 1138_bus.mat, varying between 0.043 sec and 0.41 sec. The final relative gap decreased to 5.67×10^{-17}, show the solver's effectiveness and extended computation for higher tolerances.

In the case of the matrix 662_bus.mat with size of 662×662 , convergence required 22 iterations at 10^{-1}, which increased to 500 iterations at 10^{-12}. Execution times ranged from 0.08 sec to 1.19 sec. The relative gap consistently reduced, reaching 8.89×10^{-17} at 10^{-12}.

For the matrix 685_bus.mat with size 685×685 , the solver demonstrated rapid convergence with only 7 iterations at 10^{-1}. However, iterations increased to 500 at 10^{-12}, with execution times ranging from 0.07 sec to 1.53 sec. The solver achieved a relative gap of 1.98×10^{-16}at 10^{-12}, confirming its precision.



Plot 5: Execution time and Final Relative Gap over tolerance

In the below table show the summery of all the matrix behavior based on the tolerance-

Tolerance	Matrix	Iterations	Execution Time (s)	Final Relative Gap [upto 8 precession]
[0.1, 0.01, 0.001, 0.0001, 1e-05, 1e-06, 1e-07, 1e-08, 1e-09, 1e-10, 1e-12, 1e-13]	**1138_bus.mat (1138, 1138)**	[15, 24, 33, 26, 30, 43, 61, 69, 70, 173, 500, 500]	[0.4517, 0.5001, 0.5575, 0.5241, 0.587, 0.8052, 0.9705, 0.9911, 0.9057, 1.832, 4.7307, 4.6849]	[2.58e-06, 2.4e-07, 3e-08, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
	494_bus.mat (494, 494)	[26, 63, 25, 73, 26, 24, 35, 225, 50, 65, 500, 500]	[0.0637, 0.116, 0.0873, 0.1378, 0.0949, 0.0902, 0.095, 0.2778, 0.0871, 0.0999, 0.4957, 0.5407]	[3.2e-06, 2.2e-07, 3e-08, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
	662_bus.mat (662, 662)	[22, 45, 33, 34, 28, 46, 63, 40, 53, 67, 55, 500]	[0.193, 0.2384, 0.1998, 0.207, 0.1847, 0.2495, 0.2862, 0.2026, 0.2803, 0.3363, 0.2988, 1.6149]	[1.69e-05, 2.43e-06, 1.8e-07, 2e-08, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
	685_bus.mat (685, 685)	[10, 10, 24, 36, 21, 53, 10, 20, 19, 29, 500, 500]	[0.1539, 0.1487, 0.211, 0.2746, 0.1904, 0.3173, 0.1528, 0.1809, 0.1837, 0.1975, 2.0961, 2.1065]	[2.16e-06, 1.4e-07, 4e-08, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
	Bcsstm05.mat (153, 153)	[5, 6, 7, 4, 6, 5, 5, 6, 7, 7, 4, 5]	[0.0076, 0.0057, 0.0033, 0.0051, 0.005, 0.0046, 0.0056, 0.0036, 0.0048, 0.0027, 0.0052, 0.0036]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
	bcsstm06.mat (420, 420)	[70, 91, 122, 125, 182, 144, 182, 167, 258, 209, 312, 500]	[0.0899, 0.1203, 0.1346, 0.1291, 0.1585, 0.161, 0.1809, 0.1667, 0.3887, 0.2801, 0.3449, 0.6153]	[9.04e-06, 8.6e-07, 9e-08, 1e-08, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
	bcsstm07.mat (420, 420)	[203, 282, 500, 500, 500, 500, 500, 280, 500, 500, 500, 500]	[0.3823, 0.4509, 0.5481, 0.5187, 0.612, 0.6878, 0.5902, 0.2397, 0.5029, 0.4992, 0.4382, 0.5227]	[2.85e-05, 2.86e-06, 2.8e-07, 3e-08, 0.0, 1e-08, 3.11e-06, 0.0, 7.6e-07, 5.528e-05, 0.0, 1.4e-07]
	Bcsstm09.mat (1083, 1083)	[1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]	[0.2746, 0.39, 0.3841, 0.4043, 0.3715, 0.4369, 0.6033, 0.3583, 0.2761, 0.3136, 0.3068, 0.3924]	[5.69829931, 5.69829931, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Table 3: Sensitivity Test Summery over tolerance

From the **[Table 3]** we can see that the matrices with higher condition numbers or complex structures require more iterations and time to converge, especially at lower tolerances.

4.6 Partition Number Sensitivity Test

The sensitivity of the Wolfe Projection Method to varying numbers of partitions was evaluated across multiple matrices of different sizes and characteristics. The analysis examined the effects on execution time. Partition numbers were specifically chosen based on the size and structure of

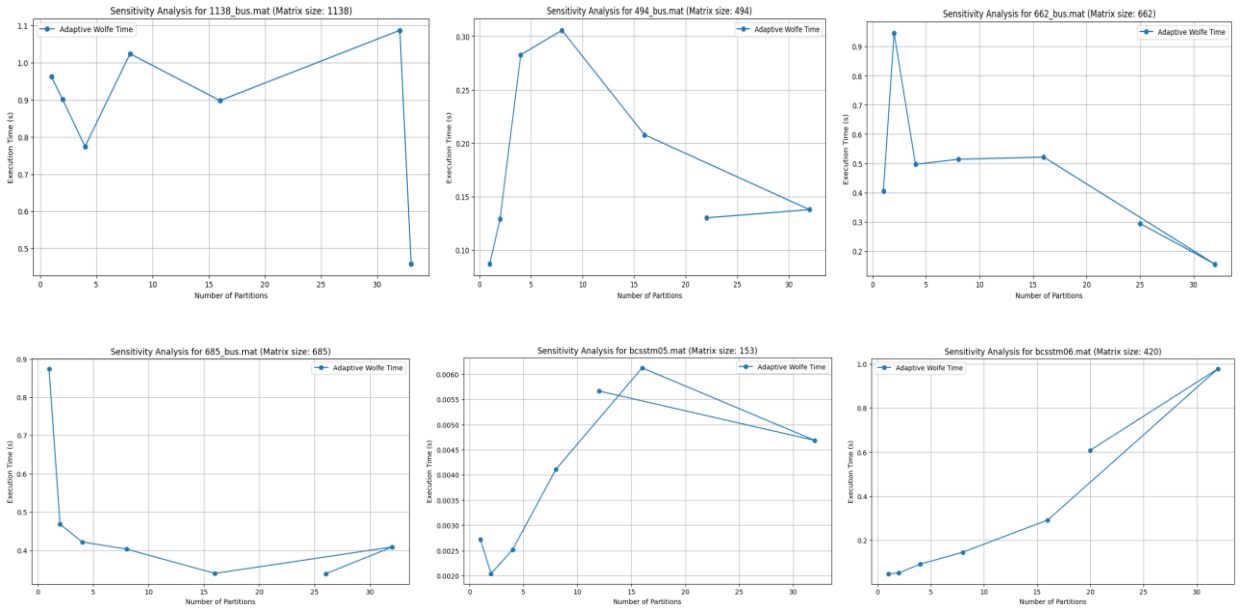
each matrix. These partitions ranged from smaller sets like [1, 2, 4, 8, 16, Isets]. Here Isets is the partition number is calculated based on the size of the matrix [Section 4.1].

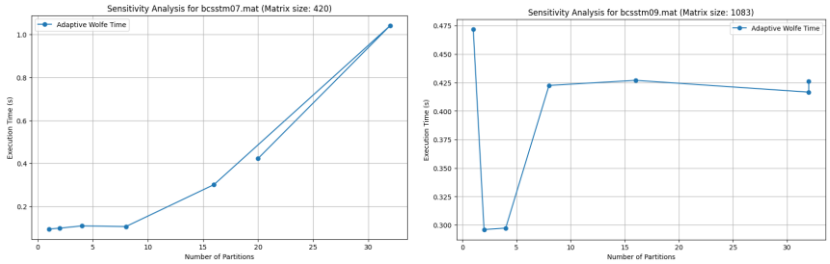
The summery table given below-

Matrix	Partition	Execution Times (s)
1138_bus.mat (1138, 1138)	[1, 2, 4, 8, 16, 32, 33]	[0.956, 0.761, 0.684, 0.837, 0.946, 0.840, 0.672]
494_bus.mat (494, 494)	[1, 2, 4, 8, 16, 32, 22]	[0.097, 0.513, 0.125, 0.171, 0.096, 0.152, 0.241]
662_bus.mat (662, 662)	[1, 2, 4, 8, 16, 32, 25]	[0.343, 0.598, 0.321, 0.288, 0.298, 0.186, 0.264]
685_bus.mat (685, 685)	[1, 2, 4, 8, 16, 32, 26]	[0.599, 0.400, 0.266, 0.348, 0.303, 0.217, 0.299]
Bcsstm05.mat (153, 153)	[1, 2, 4, 8, 16, 32, 12]	[0.002, 0.003, 0.003, 0.003, 0.003, 0.005, 0.004]
bcsstm06.mat (420, 420)	[1, 2, 4, 8, 16, 32, 20]	[0.042, 0.065, 0.086, 0.103, 0.153, 0.450, 0.250]
bcsstm07.mat (420, 420)	[1, 2, 4, 8, 16, 32, 20]	[0.054, 0.047, 0.074, 0.106, 0.712, 0.658, 0.601]
Bcsstm09.mat (1083, 1083)	[1, 2, 4, 8, 16, 32, 32]	[0.550, 0.328, 0.323, 0.281, 0.289, 0.295, 0.327]

Table 4: Summery table for sensitivity test with partition number

From **[Table 4]** and **[Plot 6]** we can see that for smaller matrices, such as bcsstm05.mat with size 153, increasing the number of partitions generally led to marginally higher execution times, with a significant increase observed for larger partition counts like 32. Similarly, in bcsstm06.mat with size 420, the execution time consistently increased with the number of partitions, indicating that partitioning overhead outweighed computational benefits for these smaller problem sizes.





Plot 6: Execution time over partition number

For medium-sized matrices, such as 494_bus.mat with size 494 and 662_bus.mat with size 662, the results were mixed. Larger partitions such as 16 or 32 often resulted in reduced execution times due to better workload distribution, although there were cases where execution times plateaued or increased slightly at higher partition counts. For example, in 662_bus.mat, the execution time was lowest at 32 partitions, but slightly higher at 25 partitions, suggesting diminishing returns from partitioning at certain thresholds.

Larger matrices, such as 1138_bus.mat with size 1138 and bcsstm09.mat with size 1083, generally benefited from increasing partition numbers, with execution times reducing significantly up to a certain threshold. For instance, in 1138_bus.mat, execution times decreased from 0.956 seconds at 1 partition to 0.672 seconds at 33 partitions. However, beyond a specific point, the benefits of additional partitions seams diminished, due to increased coordination overhead.

In conclusion, while higher partition numbers benefit larger matrices by improving scalability, smaller matrices perform better with fewer partitions to reduce unnecessary overhead.

5. Conclusion

This study explored the convergence and sensitivity of the exact step size projection method across a variety of scenarios, analyzing the impact of matrix properties, tolerance levels, and computational performance. The results demonstrated that the method is robust and capable of achieving high accuracy across diverse conditions. Convergence was consistently achieved in most cases, with the number of iterations increasing as stricter tolerances were applied. The relationship between tolerance levels and computational demands was shwon where with higher precision requiring more iterations and execution time.

The sensitivity analysis highlighted that the condition number of the matrix plays a crucial role in determining the method's efficiency. Well-conditioned matrices exhibited rapid convergence with minimal computational overhead, while ill-conditioned matrices posed significant challenges, requiring higher iterations and occasionally reaching the maximum allowed limits.

Execution time showed a clear dependence on both matrix size and the required precision. Smaller or well-behaved matrices resulted in shorter computation times, even for strict tolerances, while

larger or poorly conditioned matrices demonstrated increased computational demands. Despite these challenges, the method consistently achieved the desired accuracy, demonstrating its reliability for a broad spectrum of problems.

The exact step size projection method shown as an effective approach for solving optimization problems under varying conditions. However, its performance on ill-conditioned matrices indicates a need for further refinement. In future work we can explore adaptive strategies, preconditioning techniques, or hybrid approaches to enhance the method's scalability and efficiency, particularly for complex and computationally demanding problems.

6. Reference:

[1] "SuiteSparse Matrix Collection," Tamu.edu, 2024. https://sparse.tamu.edu/

[2] Lecture Pdf 4-Constrained optimization- Page 3,15,16

[3] A. Frangioni, B. Gendron, and E. Gorgone, "On the computational efficiency of subgradient methods: a case study with Lagrangian bounds," Mathematical Programming Computation, vol. 9, no. 4, pp. 573–604, May 2017, doi: https://doi.org/10.1007/s12532-017-0120-7.

[4] C. Michelot, "A Finite Algorithm for Finding the Projection of a Point onto the Canonical Simplex of $\mathbb{R}^n$" Journal of Optimization Theory and Applications, vol. 50, no. 1, pp. 195–200, Jan. 1986.doi: https://doi.org/10.1007/BF00938486

[5] J. Nocedal, S.J. Wright, Numerical Optimization– second edition, Springer Series in Operations Research and Financial Engineering, 2006

[6] Sébastien Bubeck (2015), "Convex Optimization: Algorithms and Complexity", Foundations and Trends® in Machine Learning: Vol. 8: No. 3-4, pp 231-357. http://dx.doi.org/10.1561/2200000050

[7] CVXOPT, "Cone Programming," CVXOPT User's Guide, [Online]. Available: https://cvxopt.org/userguide/coneprog.html. [Accessed: Dec. 29, 2024].